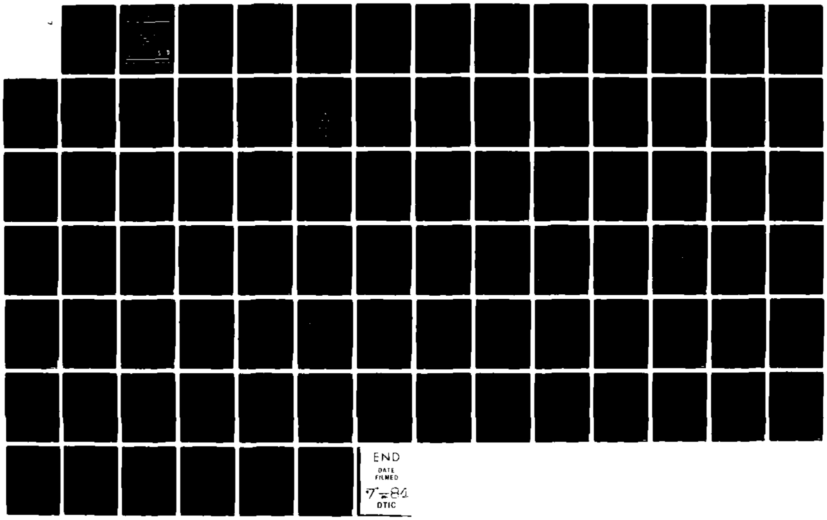
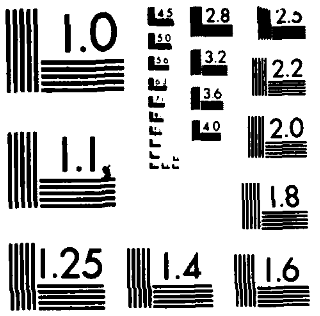



1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

ARO 18781.1-EL-S

②

FINAL REPORT

CONTRACT NO. DAAG 2982-C-0007

SPECIFICATION COMPLEXITY AND VALIDATION

MAY 4, 1984

Thomas J. McCabe

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ARO 17811-EL-S	2. GOVT ACCESSION NO. AD-A142 479	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SPECIFICATION COMPLEXITY AND VALIDATION		5. TYPE OF REPORT & PERIOD COVERED FINAL 15 MARCH 1982 14 MARCH 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) THOMAS J. MC CABE		8. CONTRACT OR GRANT NUMBER(s) DAAG29-82-C-0007
9. PERFORMING ORGANIZATION NAME AND ADDRESS MC CABE & ASSOCIATES, INC. 5501 TWIN KNOLLS ROAD, SUITE 111 COLUMBIA, MARYLAND 21045		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE May 4, 1984
		13. NUMBER OF PAGES 81
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR(S) AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

SPECIFICATION, COMPLEXITY, TESTING, ACCEPTANCE TESTING, SCENARIOS, DATA FLOW DIAGRAMS (DFD), INTEGRATION, CYCLOMATIC COMPLEXITY, MC CABE COMPLEXITY METRIC

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

THIS PAPER ADDRESSES THE INHERENT DIFFICULTIES OF LARGE-SCALE SOFTWARE DEVELOPMENT PROJECTS WITH RESPECT TO PRODUCING VALID SPECIFICATIONS AND RIGOROUS ACCEPTANCE TEST DATA. THE TECHNIQUES PRESENTED HERE USE THE INFORMATION PROVIDED BY DATA FLOW DIAGRAMS (DFDs) TO DEVELOP A MATHEMATICALLY PRECISE MODEL WHICH IDENTIFIES AND QUANTIFIES THE COMPLEXITY OF THE PROCESSES WITHIN A SYSTEM. THIS IN TURN WILL PROVIDE AN ACCURATE VALIDATION OF THE SPECIFICATIONS AGAINST USER REQUIREMENTS AS WELL AS THE INFORMATION FROM WHICH ACCEPTANCE TESTS CAN BE DERIVED.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

PREFACE

This paper addresses the inherent difficulties of large-scale software development projects with respect to producing valid specifications and rigorous acceptance test data. The techniques presented here use the information provided by Data Flow Diagrams (DFDs) to develop a mathematically precise model which identifies and quantifies the complexity of the processes within a system. This in turn will provide an accurate validation of the specifications against user requirements as well as the information from which acceptance tests can be derived. The existing state-of-the-art does not currently provide for such rigorous, early validation.

A graphic overview of this paper appears on the next page as a suggested reading guide, since the order of the different sections is not necessarily a sequential one. This overview shows at a glance how the sections are related.

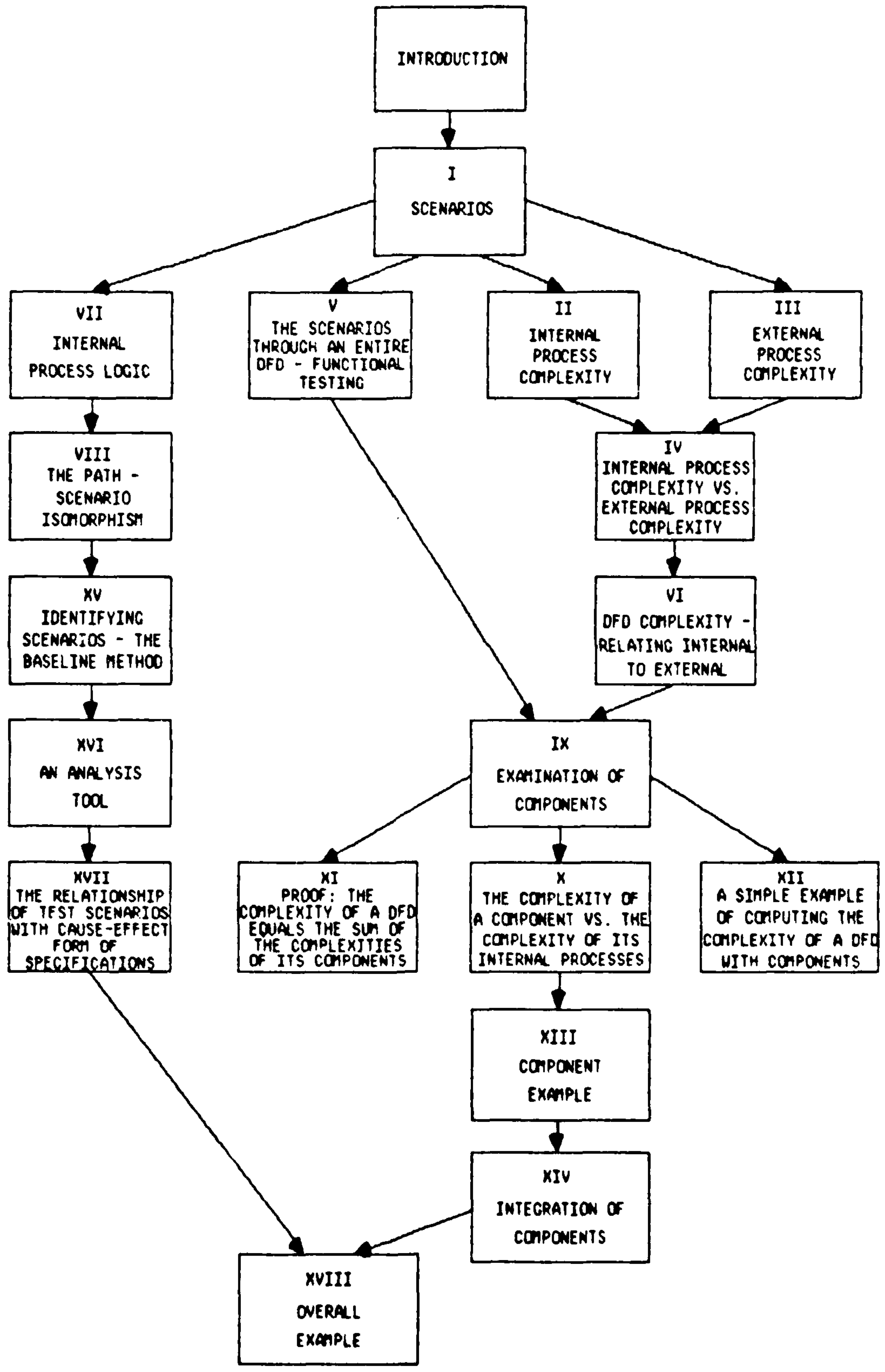
The first six sections develop the scenario concept from the identification of scenarios using DFDs to the quantification of scenarios using complexity measurement. Section VII explains the application of the scenario concept for testing while Section VIII shows the mathematical relationship of the scenario and the DFD.

Section IX identifies another aspect of the information provided by DFDs, called the component, and Section X looks at the relationship between functional processes and components. Section XI is the mathematical proof of the component-DFD relationship. Section XII explains the quantification of components, again using complexity measurement, while Section XIII gives an example. Section XIV relates the component concept to the testing environment.

The next three sections contain some how-to information -- Section XV describes a specific technique for identifying the set of scanarios within a DFD, called the "Baseline method"; Section XVI describes the Specifications Complexity Analysis Tool (SCAT) which is used for DFD complexity analysis; and Section XVII shifts the focus a bit from testing to specifications by discussing the relationship of test scenarios with "cause-effect" specifications.

Finally, Section XVIII is an overall example tying all the concepts presented in this paper together, showing the operational steps that have resulted from this research.

A GRAPHIC OVERVIEW


INTRODUCTION
I SCENARIOS
VII INTERNAL PROCESS LOGIC
V THE SCENARIOS THROUGH AN ENTIRE DFD - FUNCTIONAL TESTING
II INTERNAL PROCESS COMPLEXITY
III EXTERNAL PROCESS COMPLEXITY
VIII THE PATH - SCENARIO ISOMORPHISM
IV INTERNAL PROCESS COMPLEXITY VS. EXTERNAL PROCESS COMPLEXITY
XV IDENTIFYING SCENARIOS - THE BASELINE METHOD
VI DFD COMPLEXITY - RELATING INTERNAL TO EXTERNAL
XVI AN ANALYSIS TOOL
IX EXAMINATION OF COMPONENTS
XVII THE RELATIONSHIP OF TEST SCENARIOS WITH CAUSE-EFFECT FORM OF SPECIFICATIONS
XI PROOF: THE COMPLEXITY OF A DFD EQUALS THE SUM OF THE COMPLEXITIES OF ITS COMPONENTS
X THE COMPLEXITY OF A COMPONENT VS. THE COMPLEXITY OF ITS INTERNAL PROCESSES
XII A SIMPLE EXAMPLE OF COMPUTING THE COMPLEXITY OF A DFD WITH COMPONENTS
XIII COMPONENT EXAMPLE
XIV INTEGRATION OF COMPONENTS
XVIII OVERALL EXAMPLE

INTRODUCTION

- The Problem

Current trends in Department of Defense systems development efforts clearly indicate that an ever growing percentage of the development effort must address software. As a result, the DOD acutely experiences the same difficulties the commercial world experiences in large software development undertakings. Two of the most critical problems are the development of valid specifications and rigorous acceptance test data. This report aims itself at these problems.

A major difficulty in the development of valid specifications is that designers often do not have a clear and precise notion of what they are designing; consequently many systems are discarded or redone. For example, in two large command and control systems, 67% and 95% of their respective codes had to be re-written because of a mismatch with user requirements [1]. There are many examples of total cancellation of projects due to poor requirements - for example the $56 million UNIVAC-UNITED Airlines Reservation System, and the $217 million Advanced Logistic System [2]. Similarly, this lack of valid specifications evidences itself during the acceptance phase. A rigorous set of specification-derived test data is required, but often not present to validate functionality prior to operational release.

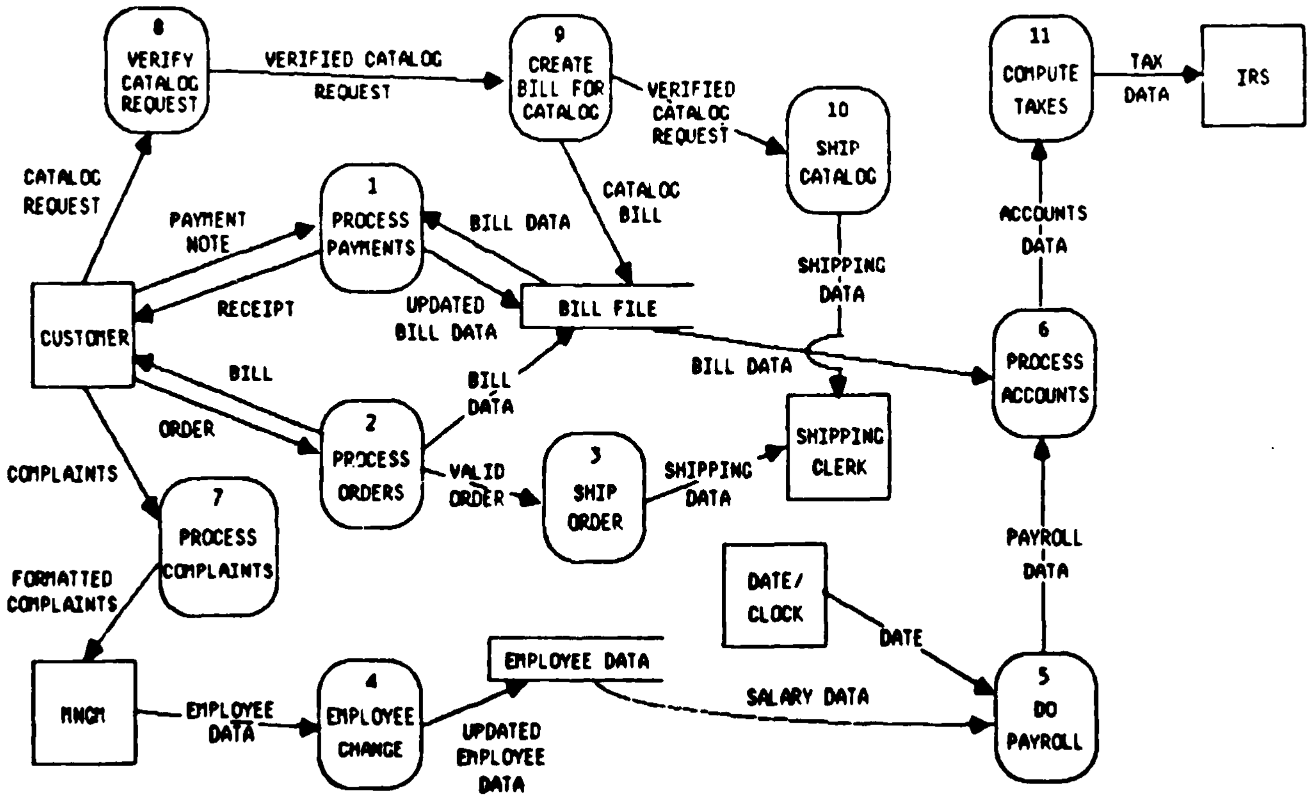
The approach presented in this paper is intended to improve the quality of specifications and test data. We will utilize the aspects of data flow diagrams to create a graph theoretic model, and apply a vector space notion to fundamentally characterize its inherent complexity. This will sensitize specifiers to the size and complexity of the specification they are creating and make explicit the testable user senarios. When successfully applied this will result in a more clear and rigorous definition of user specifications; accompanied by explicit user scenarios to walkthrough and validate the specification prior to the design stage. These same scenarios can later be applied as acceptance tests to validate the end-product before its operational use.

The two problems of involved specifications and poor test data presented here are strongly intertwined - it is impossible to derive quality test data without a valid specification, and likewise it is impossible to validate a specification without having a notion of the different user scenarios it contains. This paper attempts to solve the problem by objectively and mathematically quantifying the inherent complexity of a specification. This will allow developers to proportion their review and analysis times to the inherent complexity of a specification. The specification complexity measure will also quantify the number of user scenarios present in the specification and feed the user scenario forward to the development cycle for validation. As the cost of errors in a specification cascades through the life cycle, application of this research will reduce cost of a software project by producing more accurate specifications, as well as increasing the effectiveness of systems acceptance testing.

- Data Flow Diagrams

Our technique is based on data flow diagrams (DFDs), a graphic form of specification which is receiving wide spread application because of its facility in communicating specs to designers. DFDs, which illustrate flow of data through a system, are a product of a technique called Strucured Analysis, and typically are accompanied by a data dictionary which defines the data, and logic for each of the processes.

A DFD is an a priori network model of software behavior which shows the interrelationships among data flows, processes, data stores, and external entities. Data flows, represented by labeled arrows, are pipelines of information which flow through a system. Processes, represented by rounded rectangles, transform incoming data flows into outgoing data flows. Data stores, represented by open rectangles, serve as repositories for data. External entities, represented by squares, include all components of the environment which drive the system[3]. An example follows.



Data Flow Diagrams have gained increasing popularity as an analysis and specification tool because of their advantages over verbal forms of specifications. One advantage is that DFDs provide traceability of inputs, processes, and outputs in a form that is highly visual and easy to understand. Secondly, they provide a useful means of partitioning a system into functional components. Finally, by focusing attention on what is to be accomplished rather than how, they foster deferral of implementation details until detailed design. This enables greater understanding of the functional and process environment, and promotes better structure in the design.

- Previous Program Complexity Results

As a background, a brief review of previous complexity research will help. In 1976 McCabe published a graph-theoretic measure that quantifies a program's complexity[4]. Called the cyclomatic complexity measure, this metric is applied during the coding and unit test stage. The cyclomatic complexity measure limits the number of independent paths in a program; the result is to assure that the complexity limited programs are thoroughly unit tested and maintainable.

The McCabe metric is currently being used in the software industry to limit the complexity and guide the testing process. While this program metric has been found to be practical to apply and effective in detecting program errors, software projects would greatly benefit from the incorporation of such a metric earlier in the life cycle where the errors are higher level and more costly.

- The Challenge

With mechanisms to identify and quantify the total number of admissible flows through a system, the foundation will exist for a rigorous validation of the specifications against the user requirements. Each admissible flow can be presented in the form of a user scenario which will define an explicit interaction between the users and the system. Thus, the validation process will be driven by the complexity of the DFD. Two important benefits will be an increased user confidence in the specifications and the avoidance of misunderstandings between the users and the developers. The existing state-of-the-art does not currently provide for such a rigorous, early validation. The set of all admissible flows will also be used to form an acceptance test bed that will be fed forward to the design, coding, and integration stages to verify that the systems development conforms to its specifications.

I. SCENARIOS

INTRODUCTION

When designing, analyzing, and testing a system we are invariably interested in user interactions with the system. A user interaction will invoke a set of flows through the system. We call such a set of flows a scenario.

In general, a very large number of scenarios might exist through a system. As it would be unwieldy to test all of these, we must find a method to determine how many and which scenarios to test. We will approach this problem by showing a relationship between DFDs and graphs, and then use graph and matrix theory to define a complexity metric and system for choosing scenarios.

The distinct scenarios through a DFD are the highest level of testing. When two scenarios differ there is a major flow one has that the other doesn't - this can typically involve a global system function such as opening a major file or establishing communication with a terminal. For this reason, identifying, quantifying, and testing the distinct data flow scenarios is critical to high level functional testing.

We can identify scenarios from a white box (inside out) or black box (outside in) approach. Each of these will be discussed shortly.

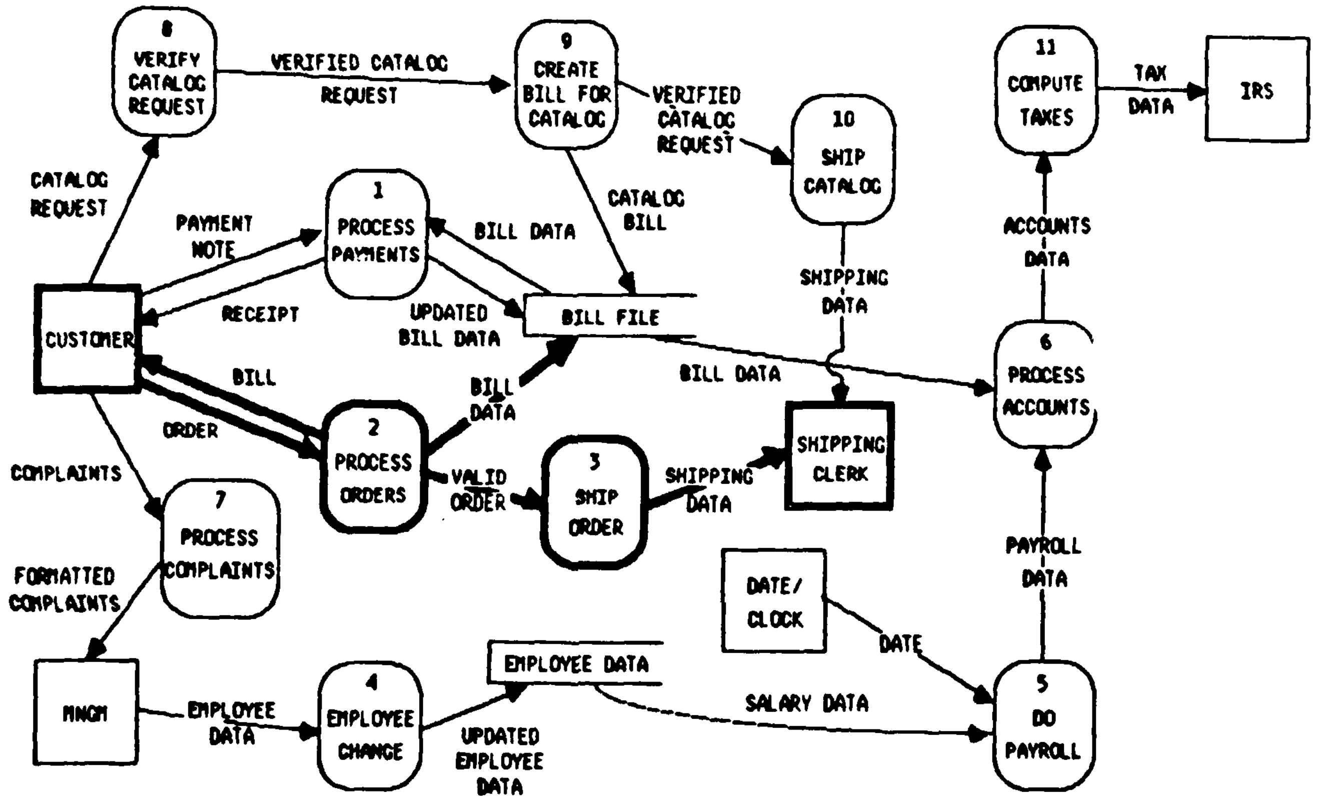
A FORMAL DEFINITION OF SCENARIO

As we now have an intuitive understanding of a scenario, let us present two technical definitions which will become useful for our later discussion.

Definition: A scenario is a physically realizable set of traversed data flows which is initiated by an external stimulus and terminated by external entities or data stores.

Definition: A scenario is an (set of) external stimulus with the data flows it induces.

To help make the idea of a scenario more clear, an example follows.



A customer files an order.
The order is processed
Bill data is sent to the bill file.
A bill is sent to the customer.
The valid order is sent out and made into a shipping order.
The shipping order is sent to the shipping clerk.

In later sections, data flow vectors will be discussed in great detail. However, for the present time, it is important just to note that two scenarios are equal if and only if their data flow vectors are equal.

II. INTERNAL PROCESS COMPLEXITY

A complexity measure will be used to limit the number of independent paths in a process so the testing will be manageable during later stages. One of the reasons for limiting independent paths, instead of a limitation based on the length of a process, is the following dilemma: a relatively short process can have an overwhelming number of paths. For example, a 50-line process consisting of 25 IF statements in sequence, will have 33.5 million potential control paths. The approach taken here is to limit the number of basis (or independent) paths that will generate all paths when taken in combination.

One definition and one theorem from graph theory are needed to develop these concepts. In this section we will treat graphs with only one connected component. For graph theory concepts and a more formal treatment of connected components, see Reference 5.

Definition 1. The cyclomatic number v(G) of a graph G with n vertices, e edges, and 1 connected component is:

$$v(G) = e - n + 1.$$

Theorem 1. In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent paths.

The application to processes will be made as follows: given a program module, associate with it a graph that has unique entry and exit nodes; each node in the graph corresponds to a block of statements where the flow is sequential and the edges represent the program's branches taken between blocks. This graph is classically known as the control graph[6]; and it is assumed that each node can be reached by the entry node and each node can reach the exit.

For example, the control graph in Figure II-1 has twelve blocks ((a) through (l)), entry and exit nodes (a) and (l), and fifteen edges.

To apply Theorem 1, the graph must be strongly connected which means that given two nodes (a) and (b), there exists a path from (a) to (b) and a path from (b) to (a). To satisfy this, we associate an additional edge with the graph which branches from the exit node (l) to the entry node (a) as shown in Figure II-2.

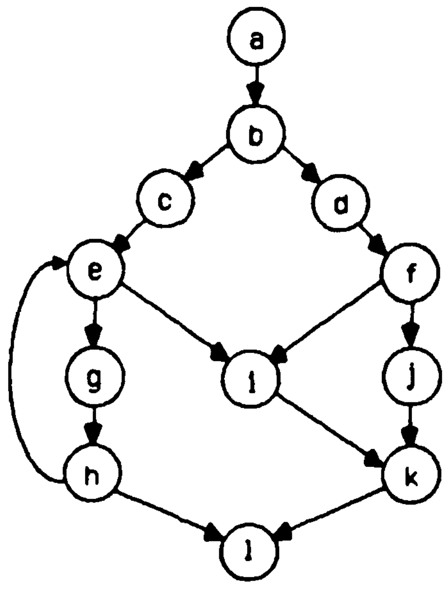


Figure II-1 Control graph G

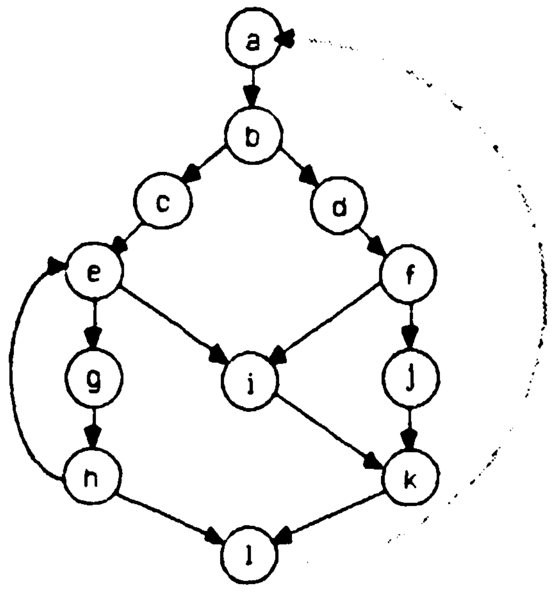


Figure II-2 Control Graph G'

Theorem 1 now applies, and it states that the maximal number of independent paths in G is 16 - 12 + 1. (G has only one connected component so we set p = 1.) The implication, therefore, is that there is a basis set of five independent paths that when taken in combination, will generate all paths. For example, the set of five paths shown below form a basis.

b_1:	abcegheikl
b_2:	abdfikl
b_3:	abceikl
b_4:	abceghl
b_5:	abdfjkl

If any arbitrary path is chosen, it should be equal to a linear combination of the basis path b_1 through b_5. For example, the path abc(egh)^{2}l is equal to $b_1 + b_4 - b_3$. To see this, it is necessary to number the edges in G (Figure II-3) and show the basis as edge vectors (Figure II-4).

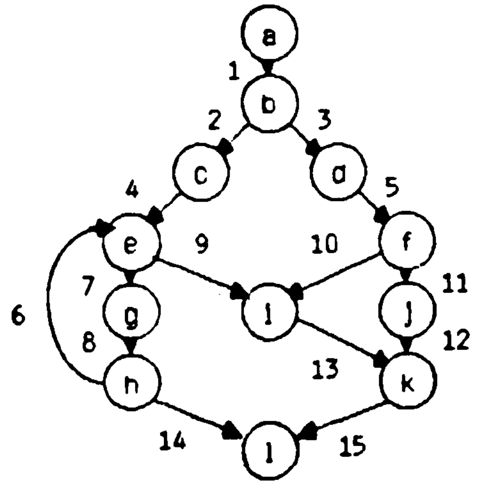


Figure II-3 Control Graph G with Numbered Paths

The path abc(egh)^{2}l is represented as the edge vector shown in Figure II-4, and it is equal to $b_1 + b_4 - b_3$ where the addition and subtraction are done component-wise.

Basis

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b1 :	1	1	0	1	0	1	1	1	1	0	0	0	1	0	1:
b2 :	1	0	1	0	1	0	0	0	0	1	0	0	1	0	1:
b3 :	1	1	0	1	0	0	0	0	1	0	0	0	1	0	1:
b4 :	1	1	0	1	0	0	1	1	0	0	0	0	0	1	0:
b5 :	1	0	1	0	1	0	0	0	0	0	1	1	0	0	1:
abc(egh)^{2}l :	1	1	0	1	0	1	2	2	0	0	0	0	0	1	0:

Figure II-4 Basis for Control Graph G

It is important to notice that Theorem 1 states that G has a basis set of size five but it does not tell us which particular set of five paths to choose. For example, the following set will also form a basis.

abdfjkl
abceikl
abdfikl
$abc(egh)^3eikl$
$abc(egh)^4l$

Note: The notation $(egh)^3$ means iterate the (egh) loop three times.

When this is applied to testing, the actual set of five paths used will be dictated by the data conditions at the various decisions in the process. The Theorem, however, guarantees that we will always be able to find a set of five that form a basis.

It should be emphasized that the process of adding the extra edge to G was performed only to make the graph strongly connected so Theorem 1 would apply. When calculating the complexity of a process or testing the process, the extra edge is not an issue, but rather it is reflected by adding 1 to the number of edges. The program complexity v, therefore, is defined as:

$$v = (e + 1) - n + 1$$

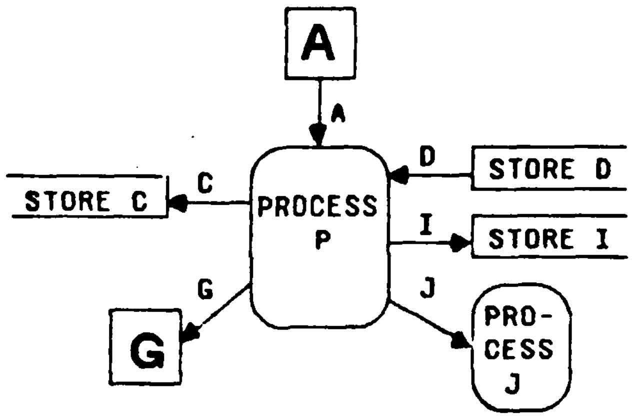
or more simply

$$v = e - n + 2.$$

III. EXTERNAL PROCESS COMPLEXITY

The preceding section dealt with the complexity inside a process. This section will address a process' external complexity - the complexity of a process' external interface to external entities, data stores, and other processes. That is, we'll look directly at the data flows that are connected to a process and deal with such issues as the number of distinct scenarios, basis scenarios, and redundant scenarios.

In order to view this we'll focus on a single process in a DFD and label its interfaces (data flows). For example the process P shown below is stimulated by the external entity A. It has data flows to the stores (files) C and I and from the store D. It also has flows to the process J and the external entity G.



The external entity (usually a computer terminal) A invokes the process P. The other flows are conditional. An example process scenario is (A,D,J); A is invoked from the terminal A, it reads data from a file D, and then invokes the process J. Notice the item (A,D,J) describes a scenario as opposed to a path. We are describing the external interface's behavior of a process rather than its internal paths.

Let the convention (A,D,J) denote the scenario that includes the data flows A,D, and J associated with the process P. We can describe P's external interface behavior by listing all such process scenarios. Let's assume with P that the following process scenarios can be realized.

(A,C,G,I,J)
(A,D,I,J)
(A,C,I,J)
(A,C,G,J)
(A,D,J)
(A,C,G,G,J)

To further analyze P's behavior let's express P's scenarios in a matrix M that spans the flows A,C,D,G,I,J as follows.

M:

$$\begin{array}{l|cccccc} & A & C & G & D & I & J \\ (A,C,G,I,J) & 1 & 1 & 1 & 0 & 1 & 1 \\ (A,D,I,J) & 1 & 0 & 0 & 1 & 1 & 1 \\ (A,C,I,J) & 1 & 1 & 0 & 0 & 1 & 1 \\ (A,C,G,J) & 1 & 1 & 1 & 0 & 0 & 1 \\ (A,D,J) & 1 & 0 & 0 & 1 & 0 & 1 \\ (A,C,G,G,J) & 1 & 1 & 2 & 0 & 0 & 1 \end{array}$$

FIGURE 4

Notice that, r(M), the rank of the matrix is 5 - there are 5 linearly independent process scenarios. The 6th scenario is a linear combination : (A,C,G,I,J) - (A,C,I,J) + (A,C,G,J).

This orientation to the outside flows of a process illustrates the application of mathematical complexity concepts to the external flows instead of internal paths. The total number of process scenarios is 6; a basis set has 5 members; the 6th scenario is formed as a linear combination of the basis set. The external complexity of the process P is defined to be the rank r of its interface matrix M - in this case 5.

Now that we can determine process scenarios, we need a method to quantify and identify which scenarios to test, so that the testing process will become less time consuming and unwieldy.

From matrix theory we know that vector spaces are spanned by a linearly independent basis set. Thus, we are not really concerned with finding distinct scenarios, but with finding linearly independent scenarios. Adding distinct scenarios can yield an enormous test set that is largely redundant - the matrix from Figure 4 is created by linear combinations of 5 basis rows. In later sections we will see 4 or 5 times as many test scenarios as in the basis set. With more complex systems, the ratio of redundant scenarios to independent scenarios can become incredibly large. Because the rank of a matrix is equal to the order of its basis set, we can quantify the dependency of a set of scenario vectors represented in matrix form. Similar to cyclomatic path complexity, the scenario complexity of a process indicates the number of basis scenarios required to span the process' test set.

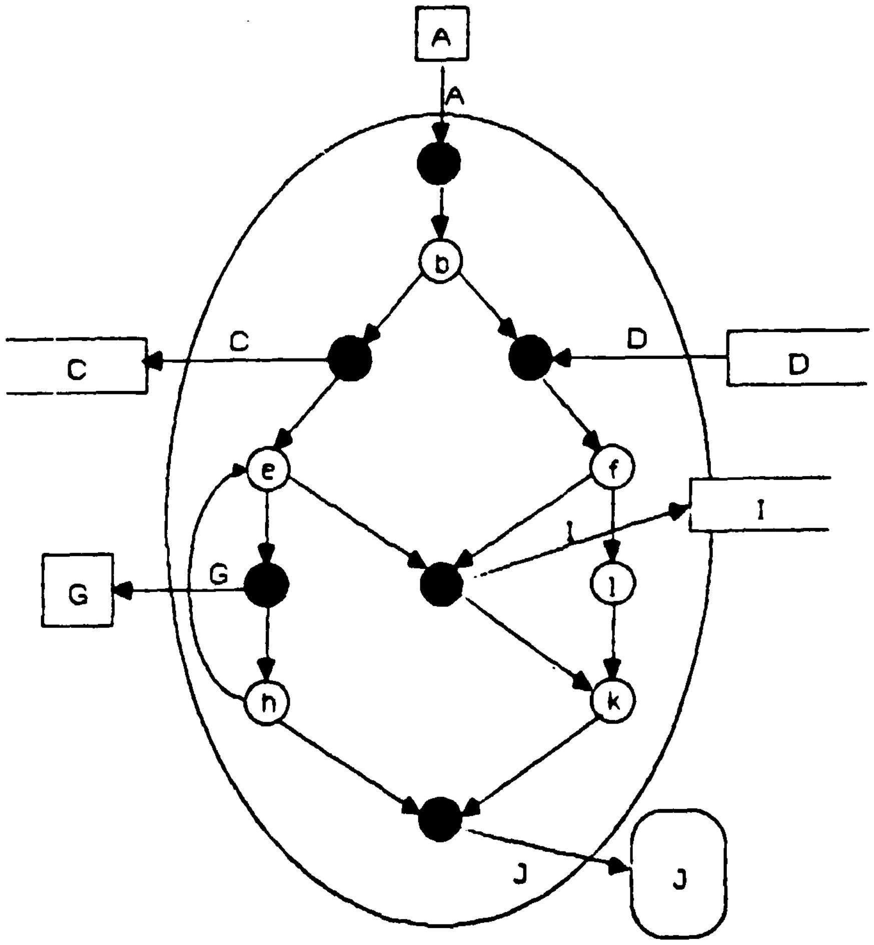
IV. INTERNAL PROCESS COMPLEXITY VS EXTERNAL PROCESS COMPLEXITY: THE RELATIONSHIPS.

For a process P we've examined its internal path complexity v(G) and its external interface complexity r(M). What's the relationship between the inside and the outside? How are v(G) and r(M) related?

The flows that come into or emanate from a process must be associated with statements (nodes) within a process that are performing interface functions: reading files, sending messages to a terminal, involving another process, etc. To highlight these statements we shade the nodes in a control graph and refer to them as black dots.

It's clear that every flow external to a process P must be connected with a black dot within P's control graph.

The diagram below shows how P's interface flows are connected to P's internal control flows. Notice the 'black dots' a,c,d,g,i, and j that connect P to the outside world.



Process P

Notice the interesting duality between the internal and external complexity. Independent external process scenarios imply independent internal paths. The independent internal paths yield independent external scenarios. In fact, we have that v(G)=r(M)=5; 5 independent paths and 5 independent scenarios.

This example is over simplified; the relationship between v(G) and r(M) will be examined in depth in later sections. The point is, however, that there is an intimate relationship between a process' internal path behavior and its external interface behavior.

V. THE SCENARIOS THROUGH AN ENTIRE DFD - FUNCTIONAL TESTING

Up to this point we have focussed on single processes. Now we turn our attention to an entire DFD.

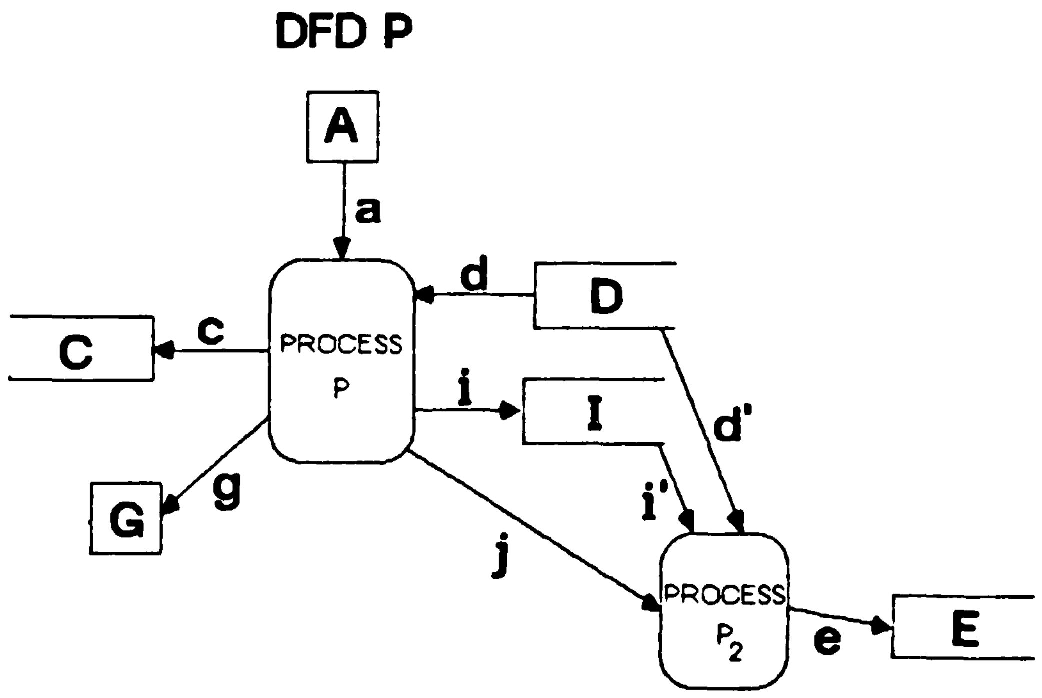
As mentioned previously, identifying, quantifying, and testing the distinct data flow scenarios is critical to high level functional testing.

When performing high-level functional testing, there are several questions to deal with:

1. how to identify and represent distinct scenarios?

2. how to determine how "good" is a set of scenarios for validation? "Good" means how complete and non-redundant is the set.

A small example will help to illustrate. We extend the example in the previous section to a complete DFD so we can talk about scenarios.



We can represent each scenario as a vector spanning the data flows a, c, d, i, j, g, d', i', and e where the entries represent the number of times the scenario traversed the respective data flow. For example the vector

$$\begin{array}{ccccccccc} a & c & d & i & j & g & d' & i' & e \\ \hline 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 2 \end{array}$$

represents the scenario aiji'ee through the DFD P.

The vector notation accurately captures the notion of a scenario. Clearly, each scenario can be represented as such a vector. Some DFDs have concurrent data flows which can flow simultaneously. This can be difficult to model. Our vector convention, however, has entries which represent the number of times a flow is traversed; this is not sequence dependent and so our vector notion can model concurrency.

Let's assume that in DFD P process P_2 either gets d' or i' and that the flow e from P_2 can repeat any number of times - e can flow 0, 1, ..., m times. The number of distinct scenarios through P is infinite; this often is the case. From a testing viewpoint a dilemma exists - we cannot test all the scenarios, we have to choose a finite number. But how many and which scenarios?

Consider the following set of scenarios, shown with its vectors in the form of a scenario test matrix, M. Recall the process P_1 has the six flows discussed in the previous section.

M:

	a	c	g	d	i	j	d'	i'	e
acgijd'e	1	1	1	0	1	1	1	0	1
adijd'e	1	0	0	1	1	1	1	0	1
acijd'e	1	1	0	0	1	1	1	0	1
acgjd'e	1	1	1	0	0	1	1	0	1
adji'e	1	0	0	1	0	1	0	1	1
acggjd'e	1	1	2	0	0	1	1	0	1

Notice the first 5 scenarios are linearly independent. No member of the first 5 is a linear combination of the others.

Notice, however, that when a sixth scenario is added, such as acggjd'e, it is a linear combination of the first 5:

acggjd'e = acgijd'e + acgjd'e - acijd'e

The goal, therefore, is not to find distinct scenarios, but to add linearly independent scenarios. Adding distinct scenarios can yield an enormous test set that is largely redundant - a test set could have 100,000 scenarios, of which 99,995 are just linear combinations of a five member basis set.

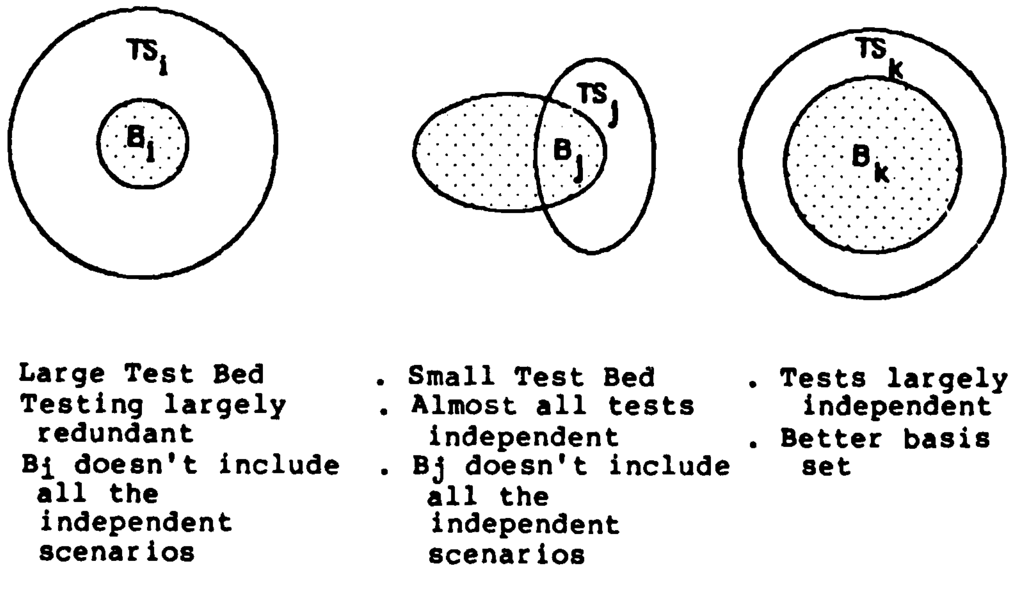
An approach is to add a distinct scenario as a row in the test matrix. If adding the scenario does not increase the rank of the matrix, discard the scenario - it's a linear combination of the existing set.

A result of matrix theory is that the rank r of a matrix M is equal to the number of linearly independent rows in M. Therefore, the rank of a scenario test matrix M is equal to the number of independent scenarios.

Given a DFD, a specific walk-through, validation, or acceptance test of the DFD defines a test data set of scenarios - TS_i. The quality of a particular test set TS_i depends on the rigor of the testing procedure and sophistication of the testing group; these factors are difficult to measure objectively.

A measurable attribute, however, is that every TS_i has a basis set of test scenarios, B_i, that will generate all of TS_i.

Our research and practical experiences show that the size of the TS_is varies dramatically dependent on the testing group and their procedures. The following pictures and comments illustrate real-world experience.



- Large Test Bed
- Testing largely redundant
- B_i doesn't include all the independent scenarios

- Small Test Bed
- Almost all tests independent
- B_j doesn't include all the independent scenarios

- Tests largely independent
- Better basis set

With the black box or external view there's good news and bad news. The good news is that every test set TS_i will have a basis set B_i. The number of tests in B_i can be computed by the rank r of the matrix M. The bad news: the quality and size of a TS_i and B_i depends on the group - it's subjective. That is, with the external view one cannot tell if the sets TS_i and B_i adequately cover the inherent complexity of the DFD.

As our goal is to define an objective complexity measure and test scenarios of a DFD, we take the following approach:

Let U_p denote the universe of test data for a DFD P.

U_p can be thought of as the union of all possible test sets:

$$U_p = \cup(TS_i)$$

Represent U_p in a matrix M_p; we use the following definition of complexity of P.

The external complexity of p is the rank r of M_p.

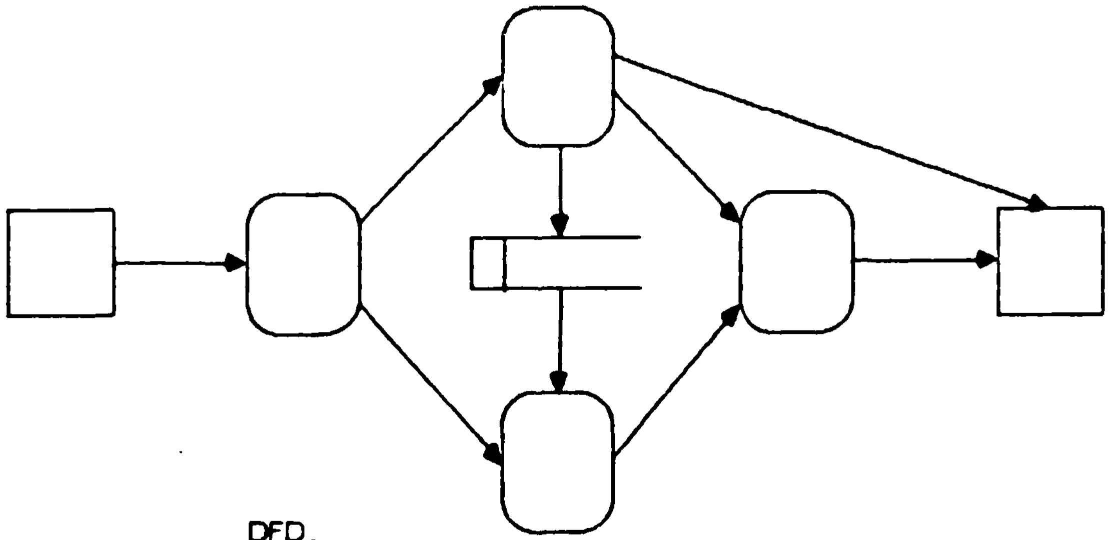
Notice that r is not dependent on an individual test set - r is solely a function of the DFD P and is therefore objective. The quantity r represents the maximum number of independent scenarios in P. We will show in subsequent sections the relationship of the external complexity r and the DFD's internal process logic complexity.

VI. DFD COMPLEXITY - RELATING INTERNAL TO EXTERNAL

In this section we will relate the DFD's internal complexity, as determined by the previous chapter, to the number of scenarios and number of basis scenarios, as evidenced externally. We will also illustrate a testing tool used in support of this research.

Three DFDs will be presented below which essentially make the points themselves. However, we need the following conventions in order to discuss them:

Let TS = the set of scenarios /TS/ = no. of distinct scenarios
B = the basis set of TS /B/ = no. of basis scenarios
r = the external complexity of the DFD



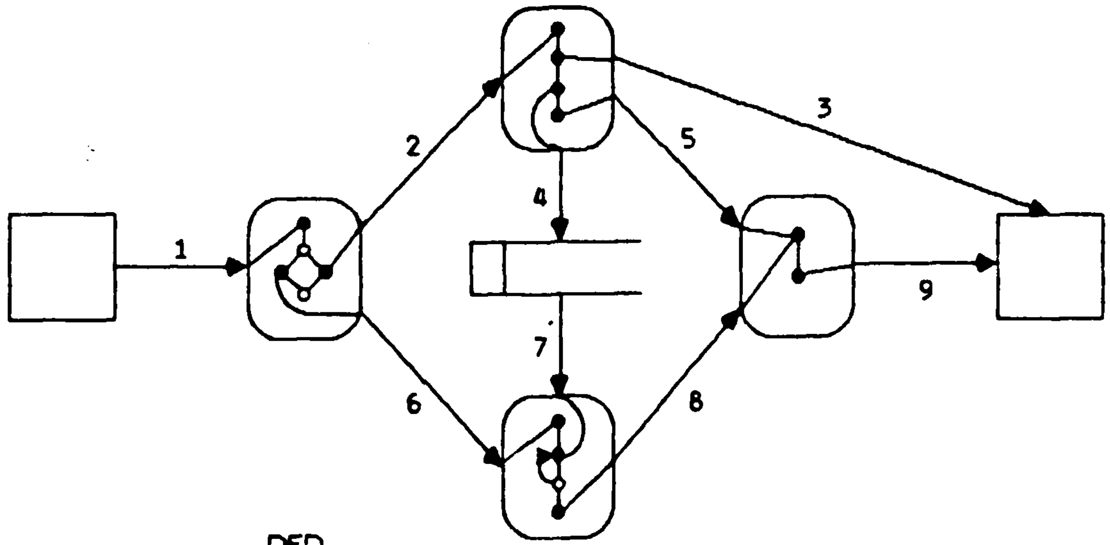
DFD_1

$v(DFD_1) = ?$

In DFD_1, since the internal process logic is not shown, TS, B, /TS/, and /B/ must all be arrived at from the outside in. Functional Test data is generated and run or walked-thru on the system - its distribution on the system results in a set of scenarios. The only way of determining the number of scenarios is empirical ... generate the test data and see what happens.

If we are given an empirically generated TS then B and /B/ can be determined objectively by computing the rank of the test set matrix. There is no way of knowing, however, if /B/ is even close to r - the external complexity of the DFD.

DFD_2 includes the graph of the internal process. A TS for DFD_2 follows.



DFD_2

$v(DFD_2) = 3$

```
Original Matrix - This is an output of the scenarios in the order originally entered.

          Data Flows
           1 2 3 4 5 6 7 8 9

    T 1    1 1 1 1 1 0 0 0 1
    e 2    1 0 0 0 0 1 1 1 1
    s 3    1 0 0 0 0 1 2 1 1
    t 4    1 0 0 0 0 1 4 1 1
      5    1 0 0 0 0 1 6 1 1
```

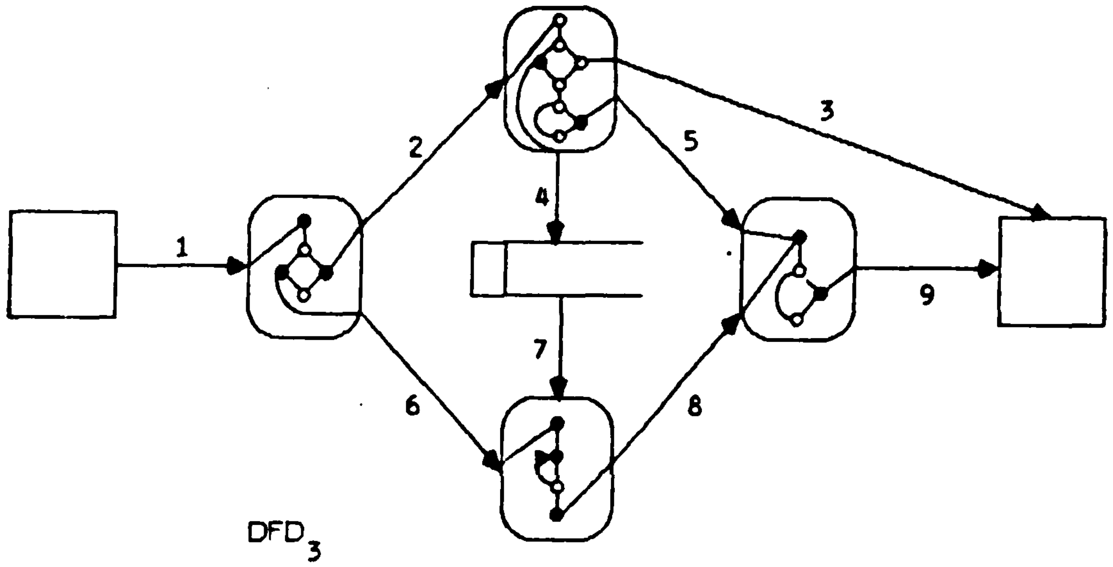
```
External Complexity = 3

Linearly Independent Test Scenarios: 2, 3, 1,
```

We have /TS/ = 5, however TS could be extended indefinitely since an infinite number of possible scenarios can be generated from the basis set of scenarios 1, 2, and 3. An additional output of the tool is the matrix shown below.

In the matrix, basis vectors will have a 1 in one column; all other columns will be zero. For instance, the first row indicates that scenario 2 is a basis scenario. To find the composition of the scenarios, follow the column entries up to a basis scenario with a 1 in that column. For example, the fourth row shows that scenario 5 is formed by 5 scenario 3s minus 4 scenario 2s.

```
Scenario Composition - This matrix indicates the composition of scenarios with respect to a basis set which spans it.

        Data Flows
          1   2   3   4   5   6   7   8   9

 T 2      1   0   0   0   0   0   0   0   0
 e 3      0   1   0   0   0   0   0   0   0
 s 4     -2   3   0   0   0   0   0   0   0
 t 5     -4   5   0   0   0   0   0   0   0
   1      0   0   1   0   0   0   0   0   0
```

The third picture is the same DFD - but the logic within the processes is more complex. With the test scenarios shown below we have /TS/ = 11 and /B/ = 6. The tool's output below identifies a basis set B.



DFD_3

$v(DFD_3) = 6$

```
Original Matrix - This is an output of the scenarios in the order originally entered.

        Data Flows
         1 2 3 4 5 6 7 8 9

T  1     1 1 1 0 0 0 0 0 0
e  2     1 1 1 0 1 0 0 0 0
s  3     1 1 1 0 1 0 0 0 1
t  4     1 1 0 1 1 0 0 0 0
   5     1 1 0 1 1 0 0 0 1
S  6     1 0 0 0 0 1 1 1 0
c  7     1 0 0 0 0 1 2 1 0
e  8     1 0 0 0 0 1 5 1 0
n  9     1 0 0 0 0 1 1 1 1
a 10     1 0 0 0 0 1 3 1 1
r 11     1 0 0 0 0 1 6 1 1
```

External Complexity = 6

Linearly Independent Test Scenarios: 1, 2, 6, 4, 7, 5,

```
Scenario Composition - This matrix indicates the composition of scenarios with respect to a basis set which spans it.

        Data Flows
         1  2  3  4  5  6  7  8  9

T  1     1  0  0  0  0  0  0  0  0
e  2     0  1  0  0  0  0  0  0  0
s  6     0  0  1  0  0  0  0  0  0
t  4     0  0  0  1  0  0  0  0  0
   7     0  0  0  0  1  0  0  0  0
S  8     0  0 -3  0  4  0  0  0  0
c  5     0  0  0  0  0  1  0  0  0
e  3     0  1  0 -1  0  1  0  0  0
n  9     0  0  1 -1  0  1  0  0  0
a 10     0  0 -1 -1  2  1  0  0  0
r 11     0  0 -4 -1  5  1  0  0  0
```

Notice with this test set that only six of the scenarios are linearly independent, so /B/ = 6. Once again, r(M) = v(DFD). The linear combinations of the basis set that generate non-basis scenarios are shown in their corresponding rows.

These DFDs suggest the following:

- Functionally testing a DFD involves identifying scenarios without looking at the internal process logic. In this case /TS/ and /B/ measure the size and quality of the externally derived test data.

- The complexity of the logic within the set of processes inside a DFD determine what /TS/ and /B/ should be! That is, the internal process logic complexity determines the DFD's external complexity.

- The same DFD with different sets of process logic yields dramatically different test sets. Furthermore, changing the internal process logic complexity changes the external DFD complexity.

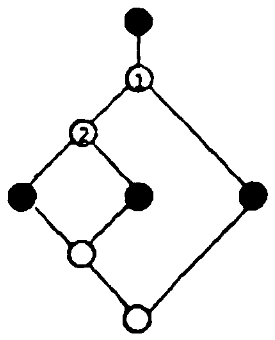
- Modification of process logic may affect r. Therefore a test group should, as a result, re-evaluate TS and B when changing process logic.

In a later section we will show that r is a function of the complexity of process logic. The point made here is that /TS/ and /B/ should be determined from the process logic and not the DFD itself. The three examples in this section were all the same DFD (they differed only by their internal process logic) - but the test data required and external complexity varied dramatically. This is why the previous computation of /B/ as the rank of a test matrix M based on empirical test data is so unsatisfactory - there is no way to know if the test data is complete.

VII. INTERNAL PROCESS LOGIC

Now we will explore in detail methods of determining data flow test scenarios and complexity by examining the logic within processes.

In previous work the notion of algorithm complexity and number of independent tests has been developed by McCabe. In short, this work quantifies the process complexity of an algorithm and has an heuristic for identifying independent paths. In several papers (e.g. the 1982 National Bureau of Standards publication)[7], it is shown that the number of independent paths in an algorithm is equal to pi + 1, where pi is the number of decision statements. For example, the graph shown below has complexity equal to 3; the nodes 1 & 2 are predicates - so the complexity is equal to 2 + 1. Notice the three independent paths through this graph that the structured testing methodology requires be tested.



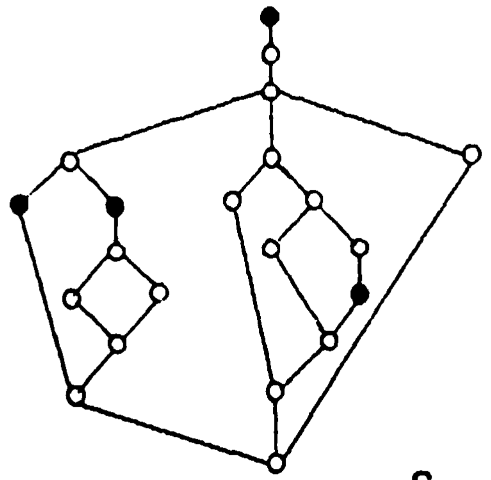
We will now explore applying the Structured Testing concepts at a higher level - that of the specification written as a data flow diagram. The first difference is that the algorithm of interest in the case of data flow specification is in high level Structured English, as opposed to a lower level compiler language. This presents no difficulties in that the complexity notions and structured testing ideas pertain to any algorithm. Even a very high level specification algorithm has a directed graph, and as such has an internal complexity and number of basis paths, even though the test data that it uses is high level functional data.

A second issue, however, requires some attention. While testing high level process logic, we must restrict our attention to only the statements and paths through a process that affect flows external to the process. There may be sections of an algorithm that represent detail that doesn't affect data flow. Our problem is to remove from the process logic the statements that don't affect flow, and concern ourselves with testing only process logic paths that affect external flow.

We will deal with this problem by adopting the following conventions. We will indicate statements which affect data flow by a black dot. All other statements will be indicated by white dots. To make this more concrete, an example is provided below.

```
Get customer-file
Add 1 to number-of-files
Select the case which applies:
        Case 1 (bill unpaid for more than 90 days)
                If quantity-owed greater than $2000,
                        Then,
                                Output lawsuit threat
                        Otherwise,
                                Output nasty bill
                                If quantity-owed less than $100,
                                        Then,
                                                Add 10% interest to
                                                  quantity-owed
                                        Otherwise,
                                                Add 20% interest to
                                                  quantity-owed
        Case 2 (bill unpaid for more than 30 days, but less than
90 days)
                If quantity owed less than $100,
                        Then,
                                Add 5% interest to quantity owed
                        Otherwise,
                                If customer-credit good,
                                        Then,
                                                Add 8% interest to
                                                  quantity-owed
                                        Otherwise,
                                                Add 10% interest to
                                                  quantity-owed
                                                Output bill
        Case 3 (bill unpaid for less than 30 days)
                Add 1% interest to quantity-owed
```

The flow graph of the example algorithm specification is shown below.

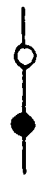


G_1

Note that all statements such as "gets" and "outputs" are indicated by black dots, while decisions and variable manipulators are indicated by white dots. Furthermore, realize that the nature of processes is such that all process graphs start with a black dot - a data flow, and have a common terminal node, which is not necessarily a black dot.

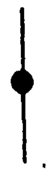
A process graph, such as the one shown in G_1, is too cluttered to readily pick out test paths, and its complexity is much higher than what we want to measure. Remember that we are only interested in scenarios which affect external flows - paths containing black dots. We will now explore a methodology for reducing the process graph to indicate only these flows. First, we will inspect some common subgraphs in process graphs, then show how to reduce them, generalize, and examine the results.

Notice the following section in G_1:

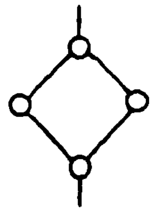


G_2

The white dot has no affect on the flow of the black dot, that is, it has no affect on external flow. We remove it, reducing G_2 to



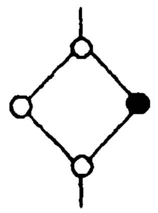
Here is another common subgraph, again taken from G_1:



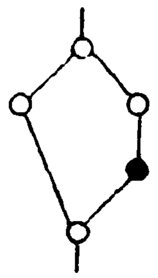
This decision block contains no black dots. As it does not affect data flow, we reduce it to an unbroken line:

To restate, this type of a decision block doesn't affect flow - we therefore remove it.

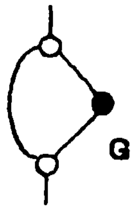
Another common decision block also appears in G_1, and is reproduced below.



Note that this is reduced from

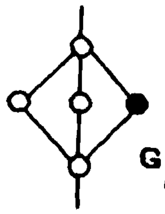


Such a decision affects black dot flow - an external flow may be triggered. Therefore, we are unable to reduce the block. However, we do remove the white dot in the branch, generating G_3.



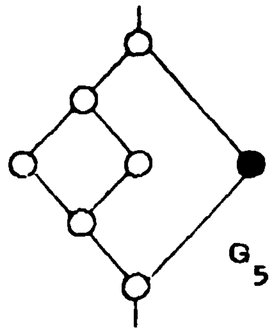
G_3

What would happen if it were



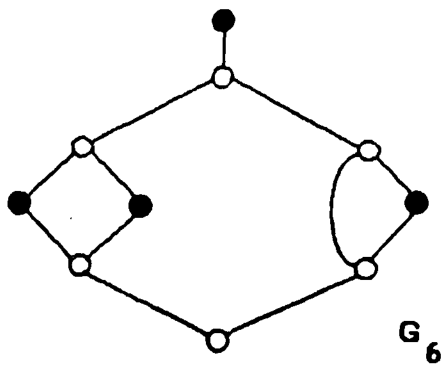
G_4.

instead? Now there are two paths through the decision (a case statement) which don't produce an external flow. We are unconcerned with which path is taken, and realize that this decision block can be reduced to G_3. Since case statements are equivalent to groups of if statements, we can redraw G_4 as



Now, it is readily apparent that this reduces to G_3.

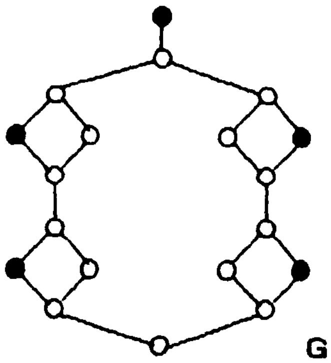
Following the steps discussed above, G_1 reduces to

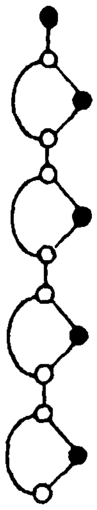


Now compare the complexities of G_1 and G_6. We find that $v(G_1) = 7$, while $v(G_6) = 4$. Three decisions in G_1 were removed. As a result, G_6 only contains decisions that affect flow. That is, it contains the minimum number of decisions necessary for all scenarios to be realizable.

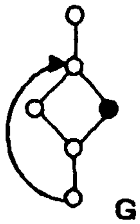
We have examined the most common subgraphs which compose process graphs and indicated how to reduce them. We will call such graphs Case 1 graphs. Sometimes, however, this is not enough to reduce a process logic graph. We will now examine how to find and reduce these cases.

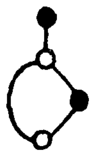
If a graph contains a white dot region - that is, an area in the graph completely bordered by white dots, further reduction is required. These white dot regions can be of two types - ones caused by decisions and ones caused by loops (while a loop is a decision, we distinguish between the two). Examples of graphs with white dot regions appear below:





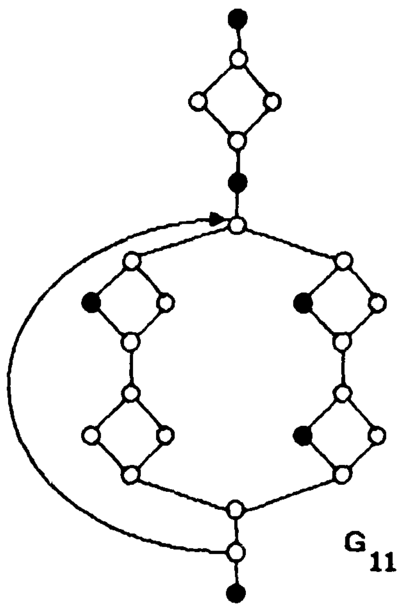
First let us consider a graph with a white dot area caused by decisions, such as G_7. We will call such a graph a Case 2 graph. Note that two paths exist through the graph which generate the same flow vectors. Also, pi + 1 = 6, but there are only 5 flows - the external complexity can't be bigger than 5! In this case, we sequentialize the decisions - generating G_8. As a result of sequentializing decisions, there are no longer paths which are distinguishable internally, but not externally, and our reduction is complete.

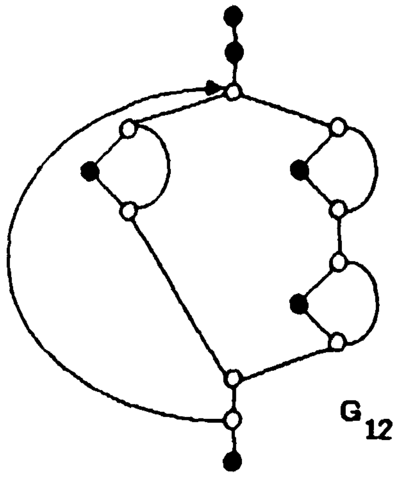


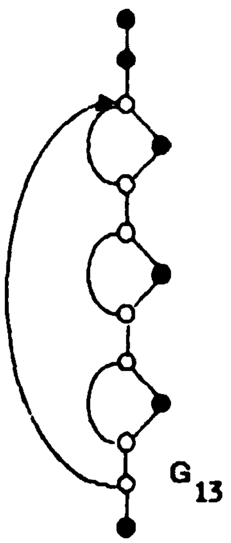


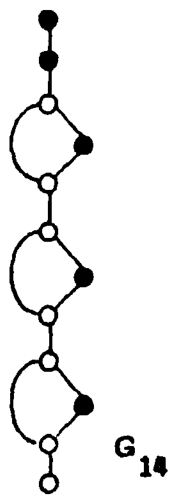
Consider G_9, however. It has a white dot region caused by a loop - now we are no longer able to sequentialize the decisions to reduce the graph. We will call such a graph a Case 3 graph. But, the loop need not be traversed to generate a set of basis scenarios! Therefore, if we simply remove it, the determination of basis scenarios will not be affected. Note that again, pi + 1 is greater than the number of flows. G_{10} is a reduction of G_9.

Now let's reduce G_{11}. First we will apply the reducing steps first discussed - generating G_{12}. A white dot region exists due to the decision structure, so we sequentialize the decisions, generating G_{13}. Nonetheless, a white dot region still remains - this time due to a loop. We remove this loop, and generate the reduced graph - G_{14}.









To summarize, perform the following steps when reducing a graph:

1) Reduce conventionally. (Case 1)
2) If a white dot region exists which is due to decision structure, sequentialize the decisions. (Case 2)
3) If a white dot region exists which is due to a loop, remove the loop. (Case 3)

As a result, we have a graph without white dot regions.

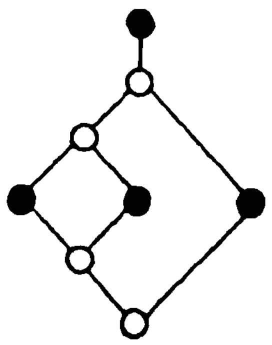
VIII. THE PATH - SCENARIO ISOMORPHISM

Now that we have examined DFDs, scenarios, and internal process logic, we will formally prove that not only does the complexity of the reduced process logic equal the complexity of the scenarios - as determined by the rank of the scenario matrix - but that the two spaces are isomorphic. We will start by reviewing some concepts, then prove isomorphism. In the process, the reasons for the reducing steps described in "Internal Process Logic" will become clear.

Viewed externally, a software system, depicted as a DFD, exhibits the effects of user interaction by flows. As shown before, we can represent the set of flows produced during an interaction as a vector. These vectors, which we call scenarios, have a basis set **B**. Let us suppose **B** contains b elements.

These vectors must be caused by something - that is, some property of the DFD. Intuitively, this is the internal process logic. Most likely, the internal process logic does more than just determine flows, so we can view the production of these flows as a subset of the actions taken by the internal logic.

We can represent each statement of the internal process logic as a node. Certain nodes must cause external flows - we will depict these as solid nodes - black dots. An example appears below.



The fact that the scenarios are generated from a graph suggests that there exists a graph, G, that only generates these scenarios. Furthermore, there should be some way to derive G from the graph of the internal process logic. We call the transformation from the internal process logic to G black dot reduction.

By showing that the method of black dot reduction creates a graph which is one-to-one and onto with the scenarios, we prove that the spaces are essentially the same - that is, that they are isomorphic. Consequently, the internal complexity, determined by pi + 1, equals the external complexity, r(M). This is a very powerful result - it establishes a fundamental link between the internal logic of a program and its external file interface. This connection between the internal structured testing of a program and its external functional testing allows us to derive external test data from the specification, evaluate its quality by r(M), and assure its rigor against the yet-to-be developed internal code.

Before we begin the proof, however, there are some mathematical notions we should review[8]:

Definition - A transform T is linear if and only if $T(ap_1$ $bp_2) = aT(p_1) + bT(p_2)$.

Definition - a linear transformation T is one-to-one if and only if $T(a_1) = T(a_2)$ implies $a_1 = a_2$.

Definition - a linear transformation T from A to B is onto if and only if for every **b** element of B there exists an **a**, element of A, such that T(**a**) = **b**.

Definition - two spaces A and B are isomorphic if and only if there exists a one-to-one linear transformation from A onto B.

For purposes of the proof we will need the following conventions:

Let S_1 be the set of all realizable scenarios. Define S to be the set of all linear combinations of the vectors of S_1.

As the number of flows in a DFD is finite, each scenario vector is finite. Clearly, S is a space and can be formed by a finite basis set. We will call S the scenario space.

Similarly let P_1 be the set of basis paths of a graph. Let P be the set of all linear combinations of P_1. Clearly, P is a vector space with basis P_1.

This chapter is designed to establish the path-scenario isomorphism theorem. It is organized in three parts: the proof of linearity, the one-to-one proof, and the onto proof. We state the theorem and proceed with the three proofs.

Fundamental Isomorphism Theorem: the space of scenarios S is isomorphic to the space P of paths.

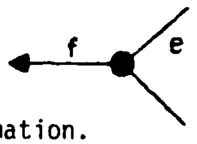
To make the transform specific we use the following conventions. For an edge e that flows into a black dot and a number c of traversals we associate with it c times the black dots flow f; i.e., $T_e(c)=cf$.

For each $p=(p_1,p_2,\ldots p_n) \in P$ define $T: P \rightarrow S$ by $T(p) = \langle T_i(p_i) \rangle$ where each e_i is the leading edge of a black dot.

Linearity Proof

Before proving T's linearity we establish this property for a simple case:

For a single edge e and its associated flow f T_e's linearity is obvious. For any two traversals c_1 and c_2 of e we have $T_e(c_1 + c_2) = (c_1 + c_2)f = c_1f + c_2f = T_e(c_1) + T_e(c_2)$.



Now the general proof follows.

Theorem - T is a linear transformation.

Proof:

Let $p= (p_1,\ldots,p_m)$ and $q = (q_1,\ldots,q_m)$ represent path vectors and $u = (u_1,\ldots,u_n)$ and $v = (v_1,\ldots,v_n)$ be their associated scenario vectors. By definition, $T(p) = u$, $T(q) = v$.

We must show $T(ap + bq) = aT(p) + bT(q)$.

By definition, $T(ap + bq) = T(a(p_1,\ldots,p_m) + b(q_1,\ldots,q_m))$
$= T(ap_1 + bq_1,\ldots,ap_m + bq_m)$

Since T is defined on black dot edges we distribute T across the m-wide path vector. This gives:

$(T_1(ap_1 + bq_1),\ldots,T_n(ap_n + bq_n))$

Since we have already shown T_i's are linear on single edges, we have the following:

$(au_1 + bv_1,\ldots,au_n + bv_n)$

Factoring out a and b yields:

$a(u_1,\ldots,u_n) + b(v_1,\ldots,v_n)$

which by definition is $au + bv$, which equals $aT(p) + bT(q)$.

One-to-one Proof

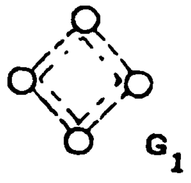
We will now state some definitions relating to process logic graphs and scenarios.

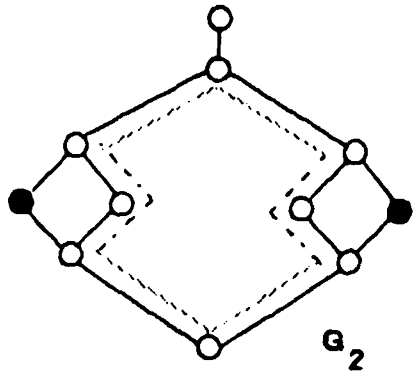
Definition - a process logic graph is one-to-one with the space of scenarios if and only if every different path through the

graph results in a distinct combination of black dots (flows).

Definition - a white dot region is a cycle bounded by white dots.

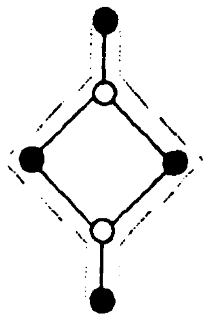
Each of the graphs below contains a white dot region.





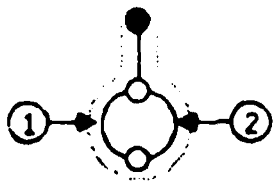
Definition - different paths span each other if they have the same initial and terminal nodes.

The two paths shown below are spanning paths.



Property - two different spanning paths are different by at least two edges.

The two paths below are spanning paths with minimal differences. They differ by two edges.



Property - if two spanning paths differ, then the graph has a decision.

Property - any edges intersected by but not common to two spanning paths bound a decision block.

The property above is obvious - as the spanning paths are different, the graph must have a decision. As the edges are not common to both paths, they must occur within a decision block. As edges define a decision block, these edges form its boundary.

Property - decision blocks in a process logic graph start and end with white dots

Theorem - Given a directed graph G, composed of black and white dots, with a unique terminal node, and a set of scenarios s_1, s_2,.... resulting from the black dots intersected by a set of paths p_1, p_2,.... The paths are one-to-one with the scenarios if and only if the graph contains no white dot regions.

Proof:

Suppose the space of scenarios is not one-to-one with the space of paths. This implies there exists two paths, p_1 and p_2, such that p_1 not equal to p_2 but s_1 equal to s_2. p_1 and p_2 paths of G implies that they have the same terminal nodes. As all process logic graphs start with a flow from an external entity, s_1 = s_2 implies p_1 and p_2 have the same initial node. Therefore p_1 and p_2 are spanning paths. This means p_1 differs from p_2 by at least two edges, and these edges define a decision block bounded by p_1 and p_2. s_1 equal to s_2 implies that none of the edges within this decision block intersect black dots. This implies there exists two different paths which intersect no black dots. This implies there exists a cycle without black dots. Therefore a white dot region exists.

Suppose there exists a white dot region. This implies that within the region there exists two different paths not intersecting black dots with the same terminal, n_e, and initial, n_i, nodes. This implies that there exists two distinct paths from the graph's initial node to n_i and from n_e to the graph's terminal node. Therefore there exists two different paths p_1 and p_2 bounding the region such that the scenarios s_1 and s_2 are the same.

Therefore, a graph is one-to-one with the scenario space if and only if no white dot regions exist.

As black dot reduction removes all white dot regions, the reduced process logic graph is one-to-one with the scenario space.

Onto Proof

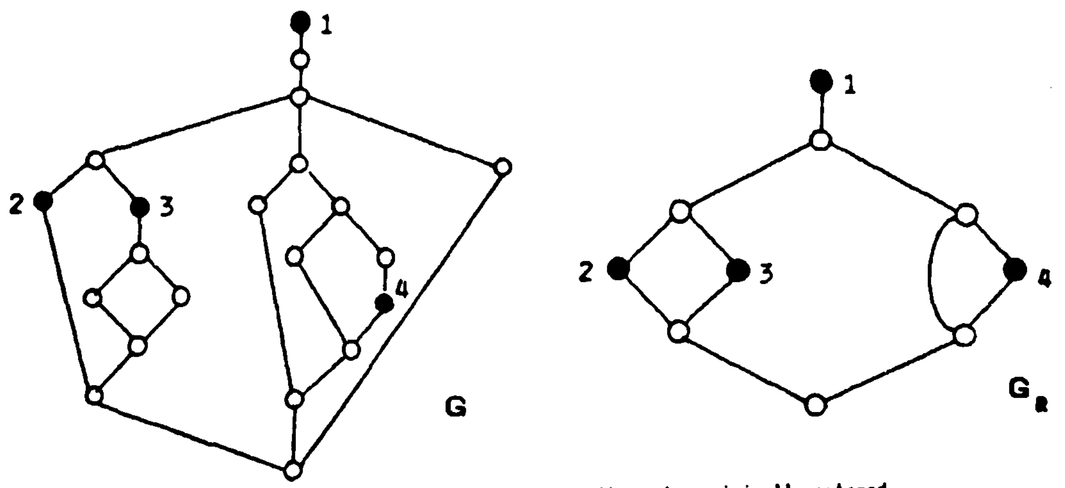
In order to show isomorphism, we still must show T maps the space of paths onto the space of scenarios. We realize that all scenarios can be determined by the intersection of paths of the process logic graph with black dots. Now, we must satisfy ourselves that every vector of the scenario space can be generated by a vector from the path space. We need to show that every scenario induced by a path in the non-reduced process logic graph is a member of the space of scenarios from the reduced graph. We will now show this to be true for each of the three reduction steps - it will then follow that the spaces are onto.

Theorem - Suppose we are given a process logic graph G with white dot regions caused only by white dot decision blocks (Case 1), and the black dot reduced graph of G, G_R. The space of scenarios generated by G can be generated by G_R.

Proof:

G_R differs from G only by white dots and decisions without black dots. Neither of these affects flows, so every scenario from G can be generated from G_R.

An example follows with two such graphs, and scenarios taken from both. Note that all scenarios from G are linear combinations of the basis scenarios from G_R.



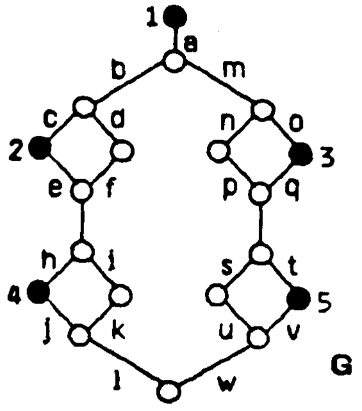
```
Original Matrix - This is an output of the scenarios in the order originally entered.

        Data Flows
         1 2 3 4

T 1      1 0 0 0  )
e 2      1 1 0 0  }  FROM G_R
s 3      1 0 1 0  )
t 4      1 0 0 1  )
  5      1 0 0 0  )
S 6      1 1 0 0  }  FROM G
c 7      1 0 1 0  )
e 8      1 0 0 1  )
External Complexity = 4

Linearly Independent Test Scenarios: 1, 3, 4, 2,
```

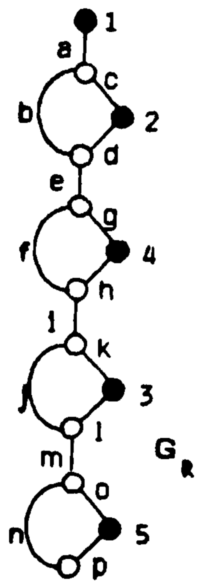
Now we must prove the onto result for the nastier Case 2 from the section on Internal Process Logic.

Theorem - Suppose we are given a process logic graph G with a white dot region caused by the decision structure (Case 2), and the black dot reduced graph of G, G_R. The space of scenarios induced by the space of paths for G can be induced by the space of paths of G_R.

Proof:

We will show this to be true for an atomic case by generating a basis set of paths from G and G_R, transforming these into scenarios, and then showing the scenarios from G to be linear combinations of those from G_R. Two graphs, G and G_R follow.





A set of basis paths for G is p_1 (a,b,c,e,g,h,j,l)
p_2 = (a,b,c,e,g,i,k,l)
p_3 (a,b,d,f,g,i,k,l)
p_4 = (a,m,n,p,r,t,v,w)
p_5 = (a,m,o,q,r,s,u,w)
p_6 = (a,m,o,q,r,t,v,w)

A set of basis paths for G_R is q_1 = (a,b,e,f,i,j,m,n)
q_2 = (a,c,d,e,f,i,j,m,n)
q_3 = (a,b,e,g,h,i,j,m,n)
q_4 = (a,b,e,f,i,k,l,m,n)
q_5 = (a,b,e,f,i,j,m,o,p)

The scenarios, i.e. the transform, of these paths appear in the tool output which follows. The list of indepenedent scenarios shows that the basis scenarios from G are linear combinations of the basis scenarios from G_R.

```
Original Matrix - This is an output of the scenarios in the order originally entered.

            Data Flows
            1 2 3 4 5

  T  1      1 0 0 0 0
  e  2      1 1 0 0 0
  s  3      1 0 1 0 0      FROM G_R
  t  4      1 0 0 1 0
     5      1 0 0 0 1
  S  6      1 0 0 0 0
  c  7      1 1 0 0 0
  e  8      1 1 1 0 0      FROM G
  n  9      1 0 1 0 0
  a 10      1 0 0 1 0
  r 11      1 0 0 1 1
  i 12      1 0 0 0 1


External Complexity = 5

Linearly Independent Test Scenarios: 1, 3, 4, 5, 2,
```

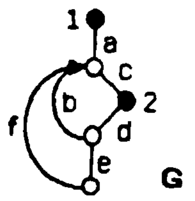
Any graph with a Case 2 white dot region will be similar to the one above. The reader should readily see that T remains onto for all reductions of a Case 2 type graph.

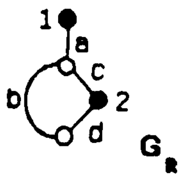
Now we must do the same for Case 3 graphs.

Theorem - Suppose we are given a process logic graph G with a white dot region caused by a loop, l_0, and the black dot reduced graph of G, G_R. The space of scenarios induced by the space of paths of G can be induced by the space of paths of G_R.

Proof:

We will conduct this proof in a similar fashion to the previous one. An atomic graph and its reduction appears below.





A set of basis paths for G is p_1 = (a,b,e)
p_2 = (a,c,d,e)
p_3 = (a,c,d,e,f,c,d,e)
A set of basis paths for G_R is q_1 = (a,b)
q_2 = (a,c,d)
$T(p_1) = (1,0)$
$T(p_2) = (1,1)$
$T(p_3) = (1,2)$
$T(q_1) = (1,0)$
$T(q_2) = (1,1)$

As $2T(q_2) - T(q_1) = (2,2) - (1,0) = (1,2) = T(p_3)$, it is clear that the basis of G is dependent upon the basis of G_R.

As before, it is clear that any graph with a Case 3 white dot region is functionally similar to the one above, so T remains onto for the reduction of a Case 3 graph.

As we have shown, all vectors from the space of scenarios from G are present in the space of scenarios from G_R, and every scenario in G_R is induced by a path in the space of paths from G_R, we have shown that T is an onto mapping.

As T is a one-to-one mapping from the space of paths to the space of scenarios, both spaces are isomorphic! Consequently, r(M) = pi + 1.

IX. AN EXAMINATION OF COMPONENTS

When examining DFDs we find a particularly important sub-structure to be quite recurrent. We will now formally examine this sub-structure of the DFD - the component. First we will state a definition of a component, then gain an intuitive notion of what a component is, relate it to what we know from graph theory, and examine consequences resulting from our definition.

The idea of a component is that of an isolated set of processes- a set that can independently operate with stimuli from only external entities and data stores. This is an important concept, as will be illustrated later.

Definition 1. A component is a set of processes and their related data flows which are surrounded by non-processes.

Again, this points out a component's isolation - its independence - from other processes within a DFD.

Enough for definitions - now we will examine several DFDs , and try to identify components within them. First look at the DFD shown below.

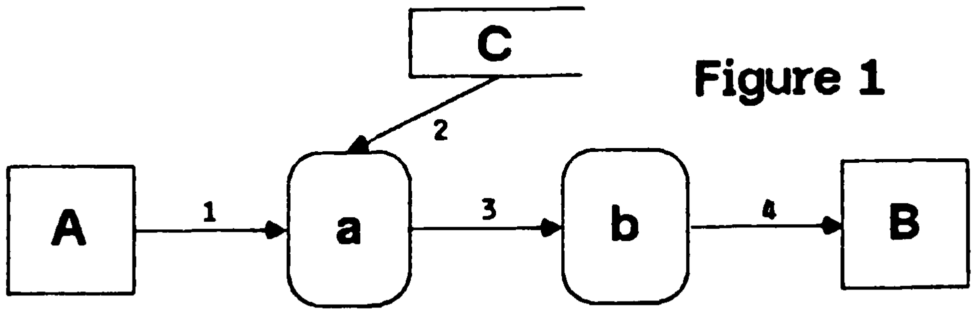


Figure 1

Process a, by itself, is not a component. While it is fed by an external entity and a data store, it also invokes process b - therefore it is not surrounded by non-processes. However, the set of processes (a,b) and their related data flows (1,2,3,4) is completely surrounded by non-processes and is therefore a component. The DFD in Figure 1 is composed of one component.

Now we will make some changes to Figure 1, and examine their effects component-wise.

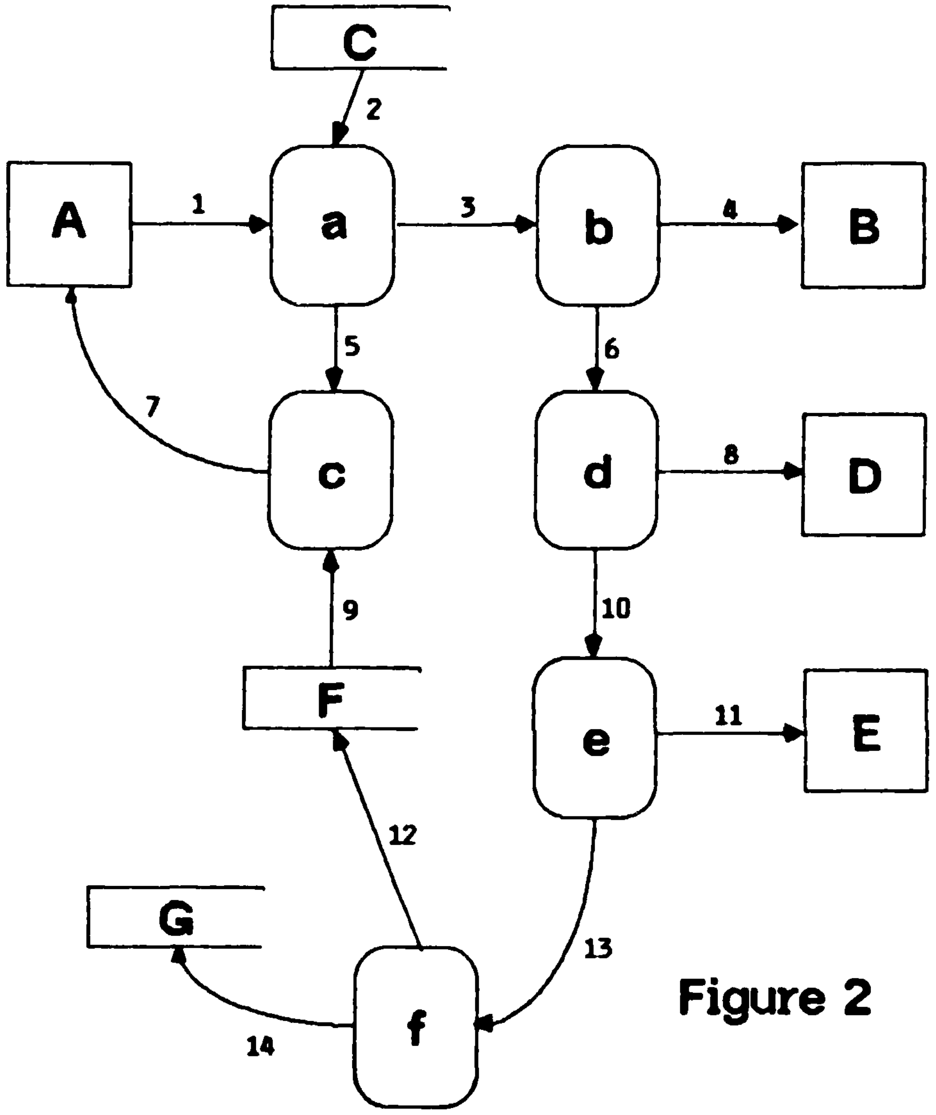


Figure 2

Our first question is, do (a,b,1,2,3,4) still compose a component? Immediately, we notice that a invokes c with data flow 5, and b invokes d with data flow 6. Now that we have ascertained that (a,b,1,2,3,4) is no longer a component, we search for new components. Following the data flows, we find no isolated sets of processes, so again we conclude that the DFD contains one component - (a,b,c,d,e,f,1,2,3,4,5,6,7,8,9,10,11,12,13,14).

We proceed to make changes on Figure 2, and examine the associated consequences.

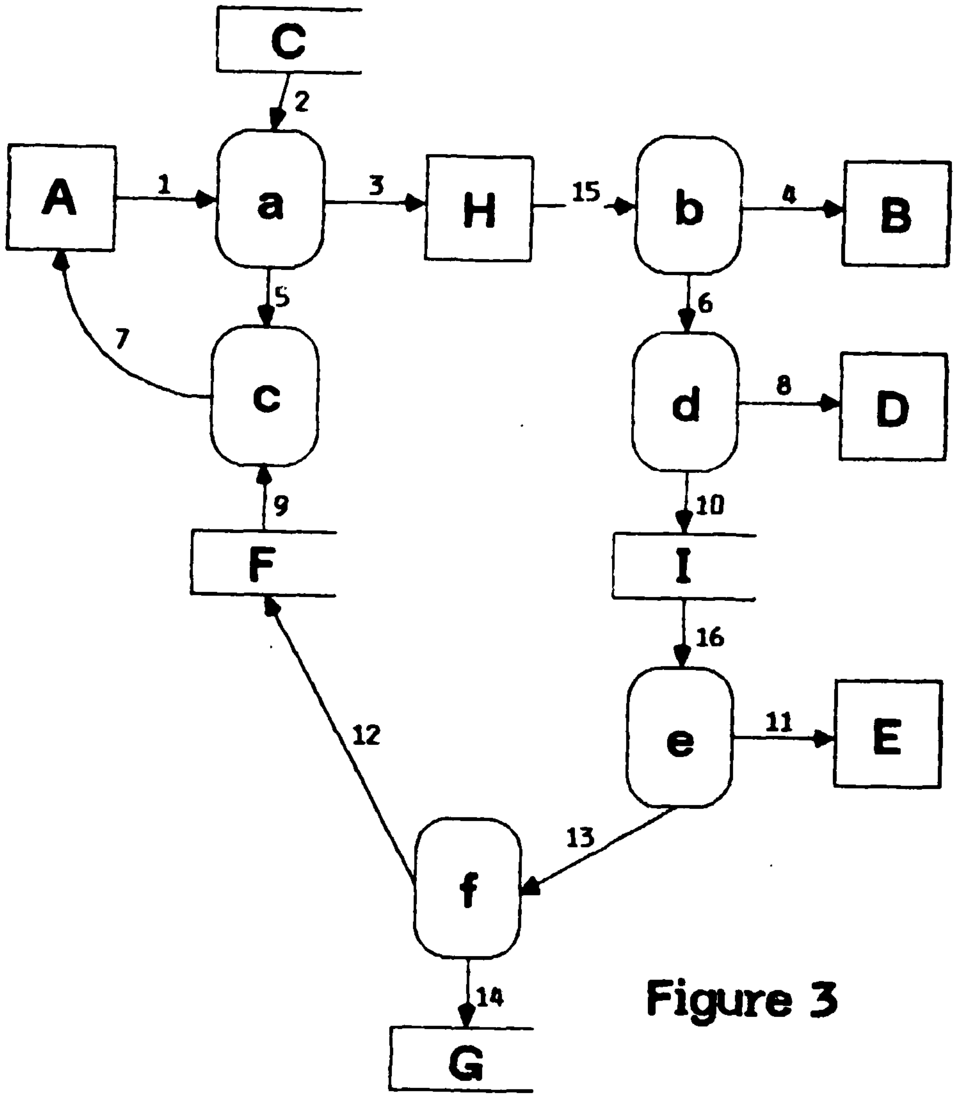


Figure 3

Now we notice radical changes - all the processes are no longer connected. First pick process a, and follow all the data flows associated with it. We find that all data flows end in non-processes except for 5, which invokes process c. However, c is connected with no processes other than a. Thus (a,c,1,2,3,5,7,9) forms a component. Repeating this procedure, we find components formed by (b,d,15,4,6,8,10) and (e,f,16,11,13,14,12). Therefore, the DFD in Figure 3 has three components.

From these exercises we notice three things: components are not difficult to identify, components can be formed by a large set of processes, and a few changes to a DFD can radically change the quantity and structure of components.

Now that we have an understanding of what components are and how to find them, we will examine an analogy between DFDs and graph theory, where the concept of a component is also used. In graph theory we define a component as the maximal set of connected nodes and edges. An important result of this definition is that two components within a graph can share no edge.

We immediately notice striking similarities between the two definitions. Processes are like nodes, data flows like edges, and data stores and external entities serve as discontinuities between flows connected to them - that is, they break a chain of connections. From this we see that another definition for a process component is the maximal set of connected processes and data flows. Furthermore, our intuitive and structural notion that no two components within a DFD can share a data flow is backed up mathematically. It follows that components are independent of each other from both a graph theory perspective and a DFD viewpoint.

Several important results follow from this examination of components. One is that components can run asynchronously within a program. Examine Fig 2 (book publishing operation, pg. 58) to see what this means in a real world situation. As an example, the preparation of vendor statements (G) can carry on independently of order verification and requisition assembly (A,B). Furthermore, these two components, (I,24,25) and (C,D,E,F,H,9,10,11,12,13,14,15,16,17,18,21,22,23) can operate simultaneously and at different rates. For machines that allow parallel operations, the independence of components becomes particularly important during translation of the DFD into a design.

Not only can components run asynchronously, but they can be tested independently as well. This is an important realization in order to optimize the testing process. Instead of developing scenarios crossing the whole DFD, only determine scenarios from a component level. Likewise, components can be programmed independently - they are an optimal unit for multiprogramming allocation.

These ideas apply in reverse also; when examining a DFD we should not find components where there is no logical concurrency. That is, a validation of a DFD can be performed by assuring the independence of identified components. This is accomplished by checking that every pair of components allows simultaneous independent execution. Even though this simultaneous execution of components may not be taken advantage of (on a simple CPU machine) the validation of at least the logical possibility of independent execution validates the component's integrity.

X. THE RELATIONSHIP BETWEEN THE COMPLEXITY OF A COMPONENT AND THE COMPLEXITY OF ITS INTERNAL PROCESSES

Now that we understand what components are, we need to establish the relationship between them and the processes which compose them. We will approach this problem from several perspectives.

First, recall the process of creating a DFD. We start with a high level view of a system, composed of a few generalized processes, and then expand these further and further. A quick example is shown below.

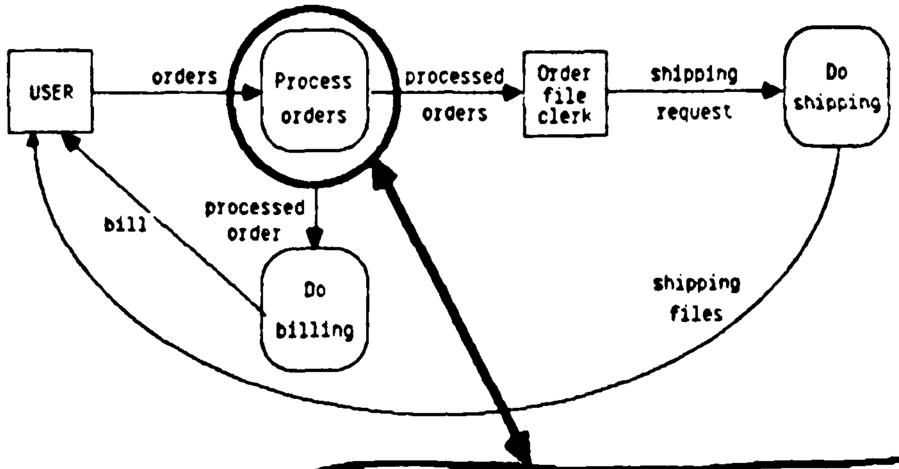


FIGURE 1

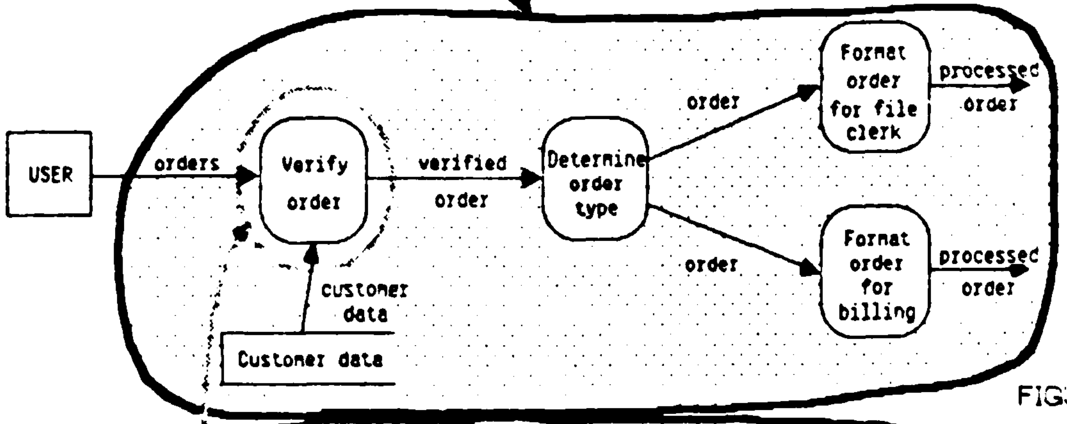


FIGURE 2

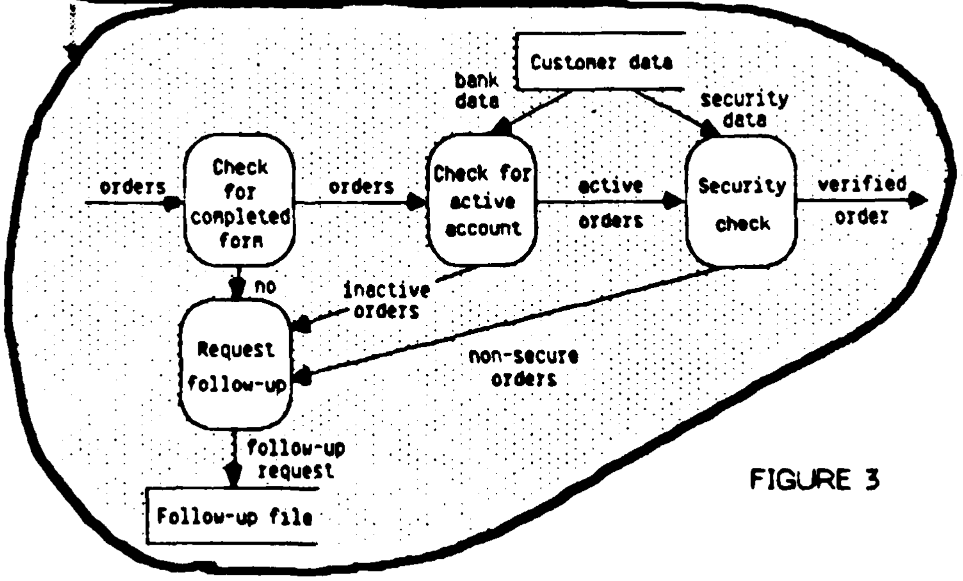


FIGURE 3

Figure 1 shows a highly generalized overview of an order receiving - shipping process. Figure 2 shows a more detailed view of the order processing section. Figure 3 shows an even more specific set of processes, all of which are encompassed by Order Verification in Figure 2. We see that such a hierarchy of DFDs extends as far as the project requires. This also illustrates the notion of lumpability - a set of connected processes can be "lumped" into one or more generalized processes.

Lumpability is an important concept to extend to components - which, by definition, are connected sets of processes. It then follows that a component can be represented as a single process - the lump of all its processes. In an earlier section we proved that the complexity of a process = $pi_p + 1$. Thinking of a component as a single process, we suppose that its complexity is $pi_c + 1$. This backs up our intuitive notion that the complexity of a component is somehow related to the complexity of its processes. A reasonable guess would be that either a) the complexity of a component is equal to the sum of the complexities of the processes which compose it, or b) the complexity of a component is equal to one plus the sum of all the pi's of each process.

Let's explore this further, examining two simple DFDs. Figure 4, shown below, is composed of one component. The internal logic of the processes, a and b, is diagrammed.

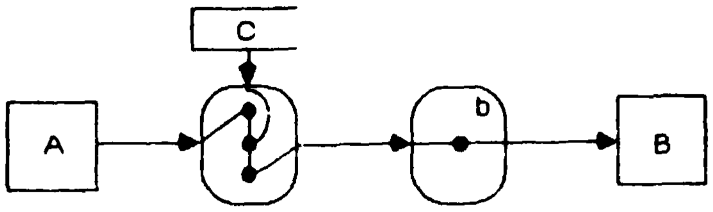


FIGURE 4

Let's compute the complexity of the component. First compute the complexity of both processes. Process a has no decision nodes, so its complexity is pi + 1 = 1. Process b, likewise, has no decision nodes, and its complexity is also one. From our previous supposition, the complexity of the component is either a) 2, or b) 1. Now, using the notion of lumpability, we generalize the component, generating Figure 5.

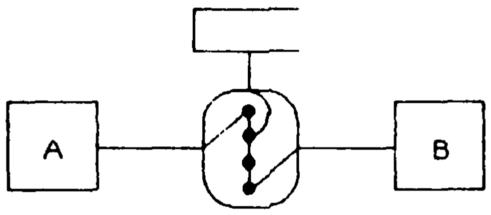


FIGURE 5

It is readily apparent that there are no decision nodes, and this processes' complexity is pi + 1 = 1. It seems that supposition b is correct. Let's examine another DFD, to see if the pattern holds.

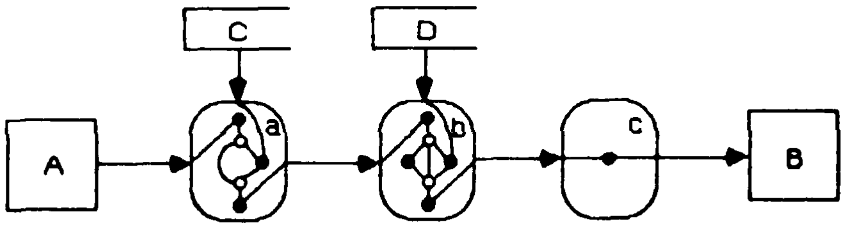


FIGURE 6

Process a has pi = 1, complexity = 2; process b has pi = 2, complexity = 3; c has pi = 0, complexity = 1. Lumping the processes together, we generate Figure 7.

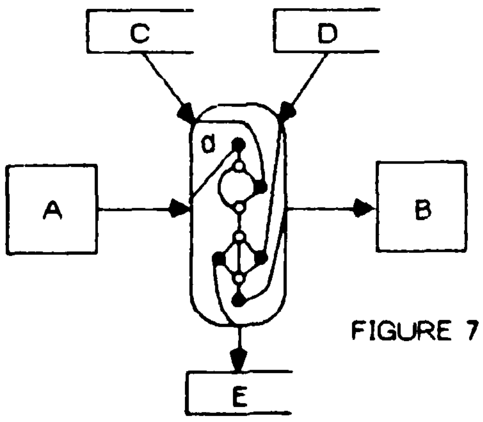


FIGURE 7

Process d has pi = 3, complexity = 4. Once again, this is what we would expect from supposition b.

It should now be apparent that the complexity of a component is equal to $\sum$(pi) + 1 where each pi is a decision in a process within the component. Let's reach an intuitive understanding of this consequence - if there are no decisions in a process, there is only one possible flow; if there is one decision, there are two possible flows, and so on. The addition of one to pi indicates that a flow always exists through the process - the flow when a decision doesn't branch. We see that pi + 1 only applies to procedures in isolation. When we look at the complexity of a component in terms of its processes, we add one to account for a flow existing through the component as a whole. We don't add one to pi for every process, as all processes in a component are connected.

XI. PROOF: THE COMPLEXITY OF A DFD IS EQUAL TO THE SUM OF COMPLEXITIES OF ITS COMPONENTS

We have shown that v(component) $= \sum pi_p + 1$. We will now prove that v(DFD) $= \sum$v(components) $= \sum (pi_c + 1)$. We will use one definition and two classical vector space theorems, shown below[9].

Definition. Let M and N be subspaces of V. We define the sum of M and N to be a subspace of V defined by $M + N = (u + v \mid u \in M, v \in N)$
If $W = M + N$ and in addition $M \cap N = \emptyset$, then we write $W = M + N$ and say that W is the direct sum of M and N.

Theorem 1. V = a direct sum of M and N if and only if we can form a basis for V by joining together a basis for M and a basis for N.

Theorem 2. V = a direct sum of M and N if and only if $\dim(V) = \dim(M) + \dim(N)$

We are given a DFD composed of n components. Remember that earlier we showed that all components are isolated from each other, and thus all flows within a component are isolated from flows existing outside it. Therefore, the component as a whole is linearly independent from all other components.

Now let us define V_{ci} as the set of all test scenarios possibly occurring within a component i. V_{ci} is composed of a basis set of linearly independent test scenarios, of rank pi + 1 (proven previously). Let $\bar{V}_{ci}$ be the set of all linear combinations of the basis scenarios. Clearly $\bar{V}_{ci}$ is closed under addition and scalar multiplication, and therefore is a subspace of $\bar{V}_{DFD}$ - the space generated by the linear combination of the scenarios within the entire DFD.

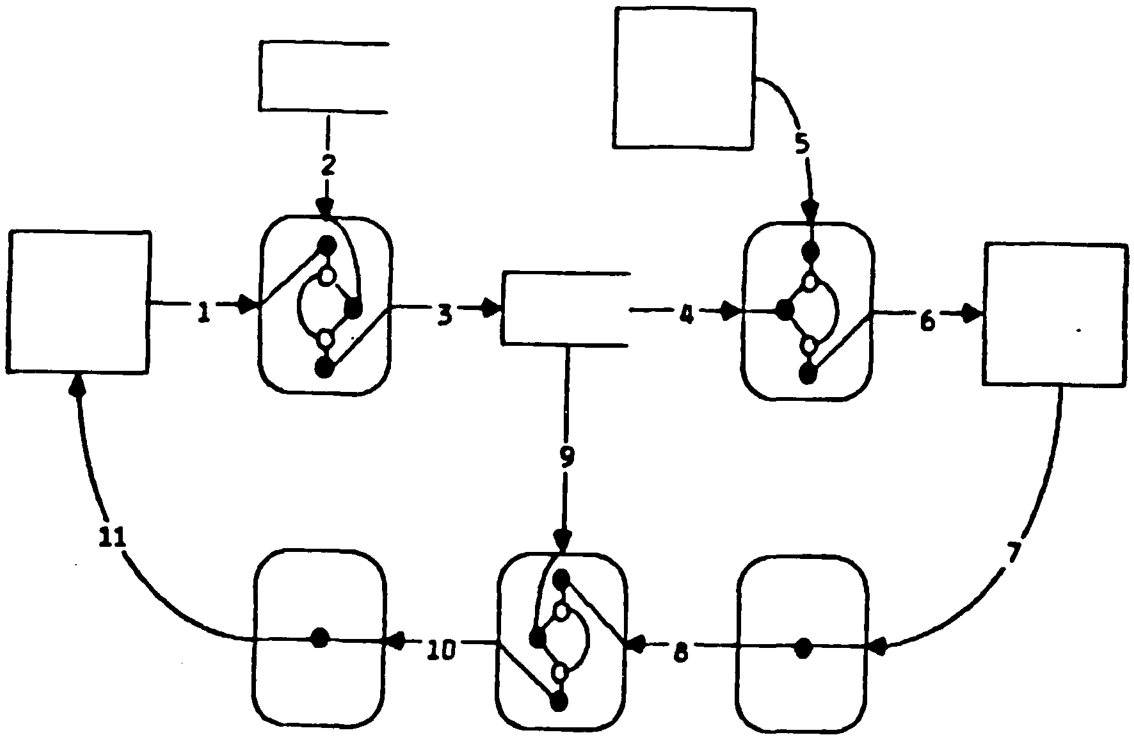
As all the subspaces - $\bar{V}_{ci}$ - are isolated from each other, their intersection is the null set. Therefore the sum of all $\bar{V}_{ci}$ is a direct_sum, which equals $\bar{V}_{DFD}$. So we have
$\bar{V}_{DFD} = \bar{V}_{c1} \oplus \bar{V}_{c2} \oplus \ldots \oplus \bar{V}_{cn}$
By Theorem 2 we have that $\dim(\bar{V}_{DFD}) = \dim(\bar{V}_{c1}) + \dim(\bar{V}_{c2}) + \ldots + \dim(\bar{V}_{cn})$
For each $\bar{V}_{cj}$ we know that $\dim(\bar{V}_{cj}) = \sum pi_j + 1$
Therefore $\dim(\bar{V}_{DFD}) = \sum (pi_c + 1)$.
This is the desired result.

Theorem 1 was not needed for the proof, it was only cited to aid our intuition. It states that we can form a basis set of scenarios for the entire DFD by joining the basis scenarios from the components.

Notice that it also follows that $r(M) = \sum (pi_c + 1)$, where M is the universal test matrix.

XII. A SIMPLE EXAMPLE OF COMPUTING COMPLEXITY OF A DFD WITH COMPONENTS

Let's examine a DFD with several components, and see if our metric $v(DFD) = \sum(pi_c + 1)$ makes sense. In addition, we should see, as proven in PROOF FOR COMPLEXITY OF A DFD EQUAL TO SUM OF COMPLEXITIES OF ITS COMPONENTS, $v(DFD) = r(M)$ (Section XI).



Simply using $v(DFD) = \sum pi_{DFD} + 1$, we get complexity to be 4. Examining a test set and its reduced matrix, shown on the next page, however, we find r(M), and thus v(DFD), = 6!

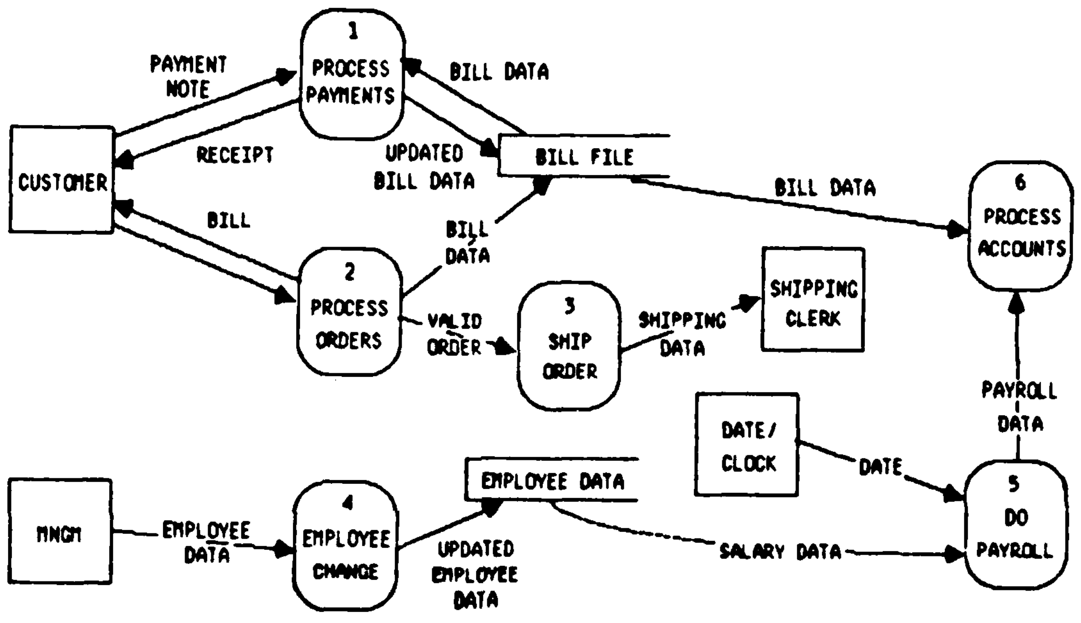
Now measure complexity by $v(DFD) = \sum(pi_c + 1)$. We find complexity to be 6, which matches r(M). Thus, our metric $v(DFD) = \sum(pi_c + 1)$ works - it matches the absolute measure r(M).

```
Original Matrix - This is an output of the scenarios in the order originally entered.

       Data Flows
        1  2  3  4  5  6  7  8  9 10 11

T  1    1  0  1  0  0  0  0  0  0  0  0
e  2    1  1  1  0  0  0  0  0  0  0  0
s  3    0  0  0  1  0  1  0  0  0  0  0
t  4    0  0  0  1  1  1  0  0  0  0  0
   5    0  0  0  0  0  0  1  1  0  1  1
S  6    0  0  0  0  0  0  1  1  1  1  1
c  7    1  0  1  1  0  1  0  0  0  0  0
e  8    1  0  1  0  0  0  1  1  1  1  1
n  9    1  1  1  1  0  1  1  1  1  1  1
a 10    2  2  2  1  1  1  1  1  1  1  1


External Complexity = 6

Linearly Independent Test Scenarios: 1, 3, 2, 4, 5, 6,
```

```
Scenario Composition - This matrix indicates the composition of scenarios with respect to a basis set which spans it.

       Data Flows
        1  2  3  4  5  6  7  8  9 10 11

T  1    1  0  0  0  0  0  0  0  0  0  0
e  3    0  1  0  0  0  0  0  0  0  0  0
s  2    0  0  1  0  0  0  0  0  0  0  0
t  4    0  0  0  1  0  0  0  0  0  0  0
   5    0  0  0  0  1  0  0  0  0  0  0
S  7    1  1  0  0  0  0  0  0  0  0  0
c  6    0  0  0  0  0  1  0  0  0  0  0
e  8    1  0  0  0  0  1  0  0  0  0  0
n  9    0  1  1  0  0  1  0  0  0  0  0
a 10    0  0  2  1  0  1  0  0  0  0  0
```

XIII. COMPONENT EXAMPLE

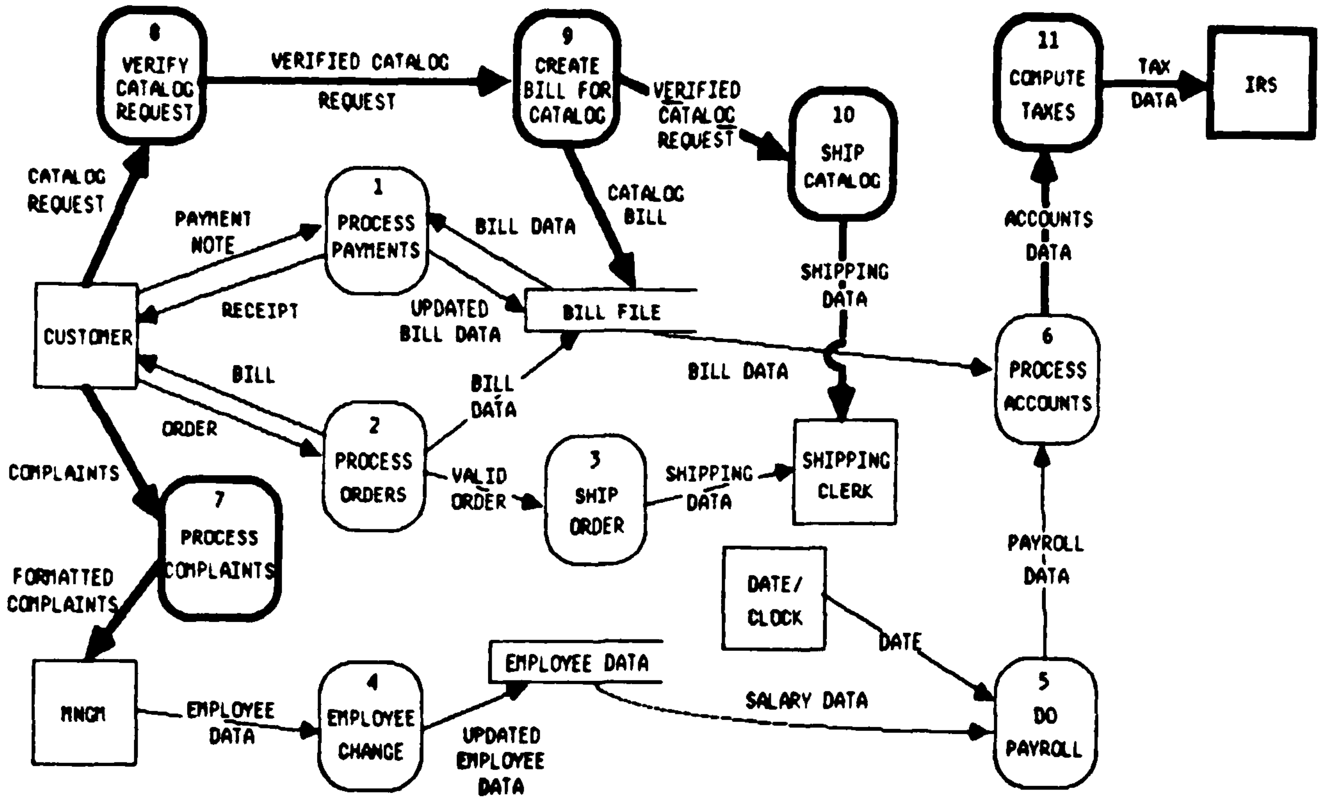
To make our discussion of components more concrete, let's examine a DFD with real processes and data flows filled in.



First identify the components. Process 1 is induced by Customer. It can read and write to the Bill File data store, and can send a receipt to Customer. Since process 1 is completely surrounded by external entities and data stores, i.e. it does not send or receive flows from other other processes, it is a component by itself. Now lets look at process 2. It is invoked by the customer as well, but it also can invoke process 3. Thus, it cannot be a component by itself. But lets examine process 2 and 3 together - they interact with no other processes. Clearly, processes 2 and 3 form a component. Now find the other two components.

To illustrate the relationship between the number of components and complexity, assume each component has complexity one. Then, as $v(DFD) = \sum v(C)$, we get $v(DFD_1) = 4$.

Now let's make the system more involved.

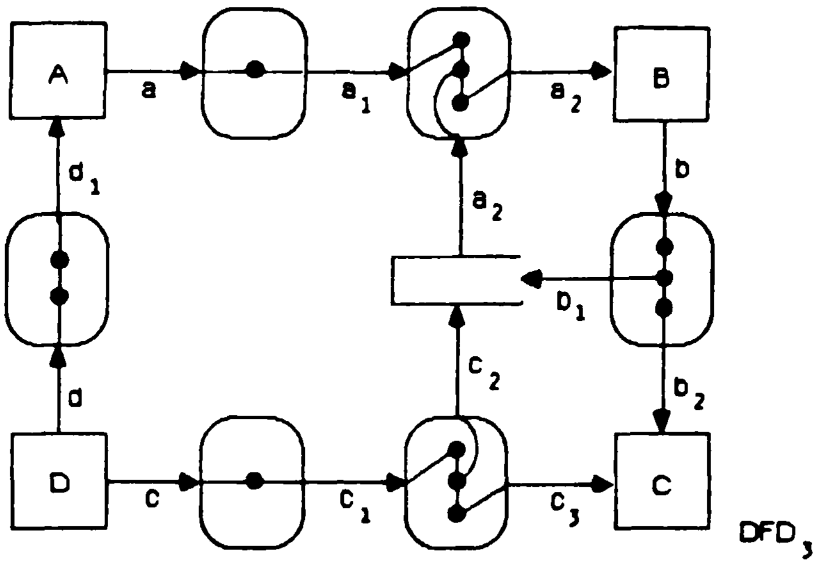


Visually the DFD appears much more complex - let's see if the quantification supports our intuition. Count the number of components. There are six - two more have been added. If we again assume unit complexity, the DFD's complexity is six. Thus, the numbers reinforce our intuition - adding components increases complexity.

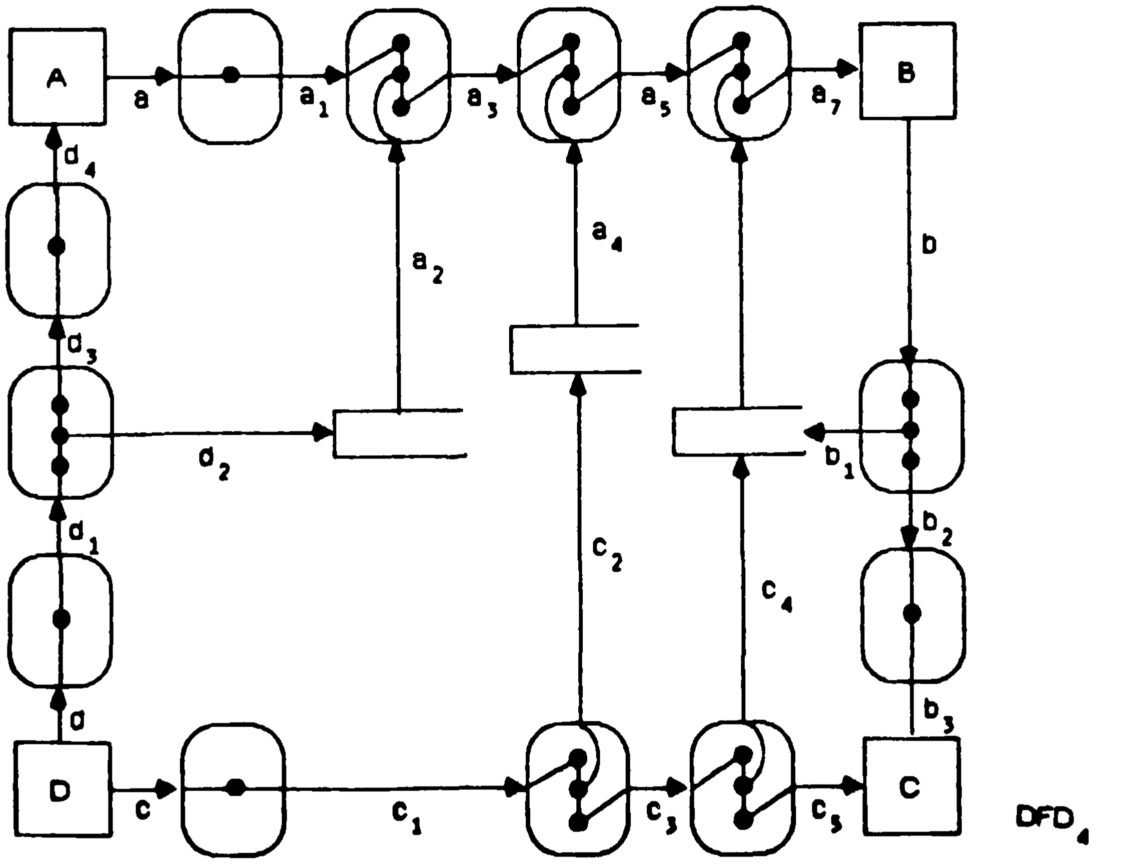
We have shown $v(DFD) = \sum v(C) = \sum(pi + 1)$. Suppose pi = 0. Then, $v(DFD) = n(C)$. As pi isn't always 0, the number of components forms a lower bound for the complexity of a DFD. Furthermore, the complexity of a DFD increases by at least one for every component added.

Let's examine the nature of the added processes to see how they affect the DFD. Again, we'll assume unit complexity - that is pi = 0 - for each process. Look at process 11 - it will always execute after process 6, so adding it does not create a new linearly independent scenario - it won't increase complexity. Process 7, however, is different. It forms a component and thus can operate independently of the other components. For example, a customer can make a complaint and request a catalog while management makes an employee change. Process 7 is clearly independent of the other components, and any scenario (as we assumed pi = 0, the scenario) through it adds to complexity. Therefore, it adds to complexity.

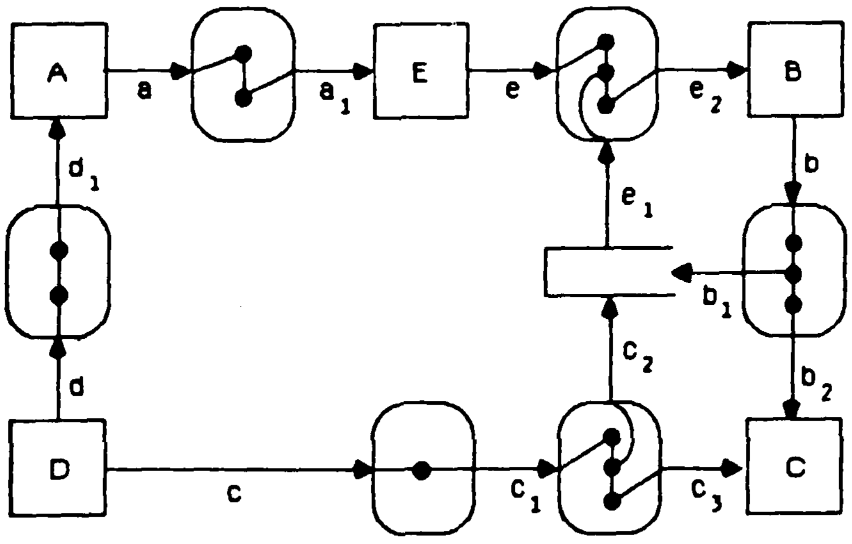
To further emphasize that adding components increases complexity, consider the DFDs below. Process logic is filled in.



DFD_3 has four components. Each has unit complexity - therefore the DFDs complexity is 4. By their nature, each component can operate independently of the others. For example, external entity A can send flow a, which will induce flows a_1 through a_3 at the same time that flows c_1 through c_4 are stimulated. Note that all components start, as all must, with a flow from an external entity.



Now let's look at DFD_4 - a modification of DFD_3. Some new data stores and processes have been added. Let's see how these affect the complexity of the DFD. If A sends data flow a, then it will also send a_1 through a_7. More data stores are accessed, but the complexity has not changed. Likewise, while the other components also access more stores, their complexity has not increased. We see that all components have complexity 1. As there are 4 components, the complexity of this DFD is also 4.



DFD_5

DFD_5 is another modification of DFD_3. The only change is that an external entity has been added which splits the upper component. Let's see how this affects complexity. As before, if flow a goes then a_1 also flows, but now the flow is stopped by an external entity, E. This external entity can send flow e, which will induce e_1 and e_2 - this forms a new component. As with DFD_3, external entities B,C, and D all invoke their own components. Count the components - there are 5. We see that each has unit complexity, so the DFDs complexity is 5. While DFD_4 is busier than DFD_5 and has more symbols, it is not as complex - DFD_5 has more independent scenarios. The increase in complexity was caused by the addition of another independent component.

XIV. INTEGRATION OF COMPONENTS

Up to this point our testing approach has concentrated upon the number of scenarios within a component. We will now look at the issue of integrating components. It is interesting that our ideas will be related to the classical notation of top-down integration of a design structure. The top-down design integration ideas have value because of two points:

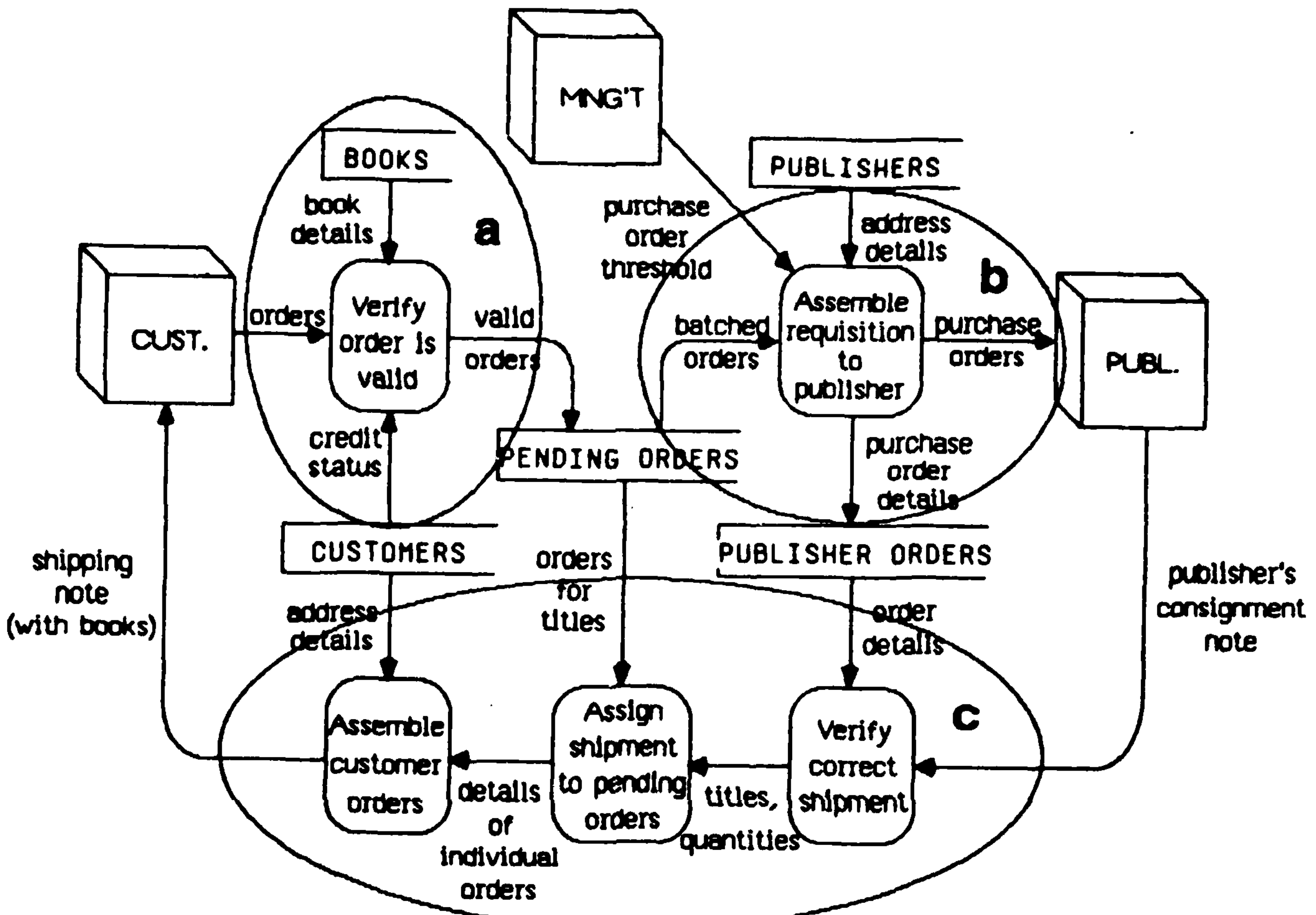
(a) the nature of the integration avoids using drivers with artificial test data

(b) the integration strategy allows the unit test data to be real system test data.

The component integration approach we will describe also fits both of these points. It's unrealistic to insist that our approach be completely done on every system, but when it's violated artificial test data will be introduced that will typically lead to integration problems where two components work individually but don't work together.

Integration testing involves, as the name suggests, testing the way components integrate. The technique is best illustrated with an example. We will revisit our book ordering DFD shown below. The components a, b & c are circled. The order of components suggests testing the scenarios within the order verification component first. For example, its complexity could be 3, resulting in 3 distinct sets of data being deposited in the pending orders data store. Next we will test component b using the data already placed in the pending orders data store. Suppose that the complexity of component b is greater than that of component a, say 4. Now we are not supplied with enough data in the data store, so we execute component a once more so that enough test data is present. In this manner, we are not producing an artificial test; all the data run through component b is that which is produced in a normal execution. Now we have a set of data produced by component b in the publisher orders data store. We proceed to test component c. Note that again we are using realizable data sets. If the complexity of component c is greater than that of component b and component a, in this case 5 or more, we will again have to fill the data stores by executing components a & b. By testing in this manner, all the tests conducted use realistic data that was produced by valid user realizable scenarios.

It makes little sense for a component of very high complexity to feed a component of low complexity which doesn't utilize all the data fed to it. Considering the relationship between data generated and used, and that of data used and generated leads us to an intuitively pleasing law:



BOOK PUBLISHING DFD[10]

The Law of Conservation of Data:

1) All data drawn from a data store must have been generated by execution of a scenario. (what goes out must have come in)
2) All data placed in a data store must be used in execution of a test scenario. (what goes in must go out)

Following this law leads to testing only valid user realizable scenarios.

Now let us see how applying the Law of Conservation of Data affects the testing situation. We will discuss some examples and summarize the results.

Once again let us look at the book publishing DFD. Let's say that the first component will not allow an invalid order, such as one for a nonexistent book, to be placed in the pending orders data store. It therefore makes little sense to place such an order in the pending orders data store when testing the second component. How the system handles the error is insignificant, as such an error will not occur in natural operation of the system. Rather, the error is not a user realizable error. Nevertheless, if the Law of Conservation of Data is not followed this type of error may occur - either the tester may be unaware of the restriction (especially if different groups test different components), or data could be improperly entered into the data store.

We see that it makes sense to only test situations which will occur in natural operation of a system. Using artificial test data within a component will introduce errors when integrating components. Following the Law of Conservation of Data, which enforces our idea of integration, prevents such data from entering a system. One might ask, "What if a process stores data in a data store, and it's not read again - then how can the Law of Conservation of Data be followed?" Such a question, however, is meaningless for real situations. A system should never create data it will not ultimately use. Such a function is extraneous and unnecessary. A system must ultimately use all the data it creates - if not, why create it?

Notice, on the other hand, that in the book publishing DFD data is drawn from a books file, while none is put in. Once again, one can ask about the Law of Conservation of Data, as this clearly violates the second clause. The answer is that the data was placed in that file in a different section of the system. The Law of Conservation of Data applies at a global, not local level. Once again, the Law of Conservation of Data applies to any proper system.

XV. IDENTIFYING SCENARIOS: THE BASELINE METHOD

The technique described here gives a specific methodology to identify a set of scenarios. When applied, this results in a set of scenarios equal in number to the complexity of the DFD as defined by pi + 1. The technique is called the baseline method, and requires black dot reduced process logic. This method is called the baseline method due to its similarity to a method for finding basis paths through a program, as detailed in McCabe's Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric[11].

The first step is to pick a functional "baseline" scenario through the DFD which represents a legitimate function and not just an error scenario. The selection of this baseline scenario is somewhat arbitrary. Realize that this baseline scenario represents a sequence of decisions taken in a particular way.

The next step is to identify the second scenario by locating the first decision in the baseline scenario and flipping its result while simultaneously holding the maximum number of the original baseline decisions the same as on the baseline scenario. This is likely to produce a second scenario which is minimally different from the baseline scenario.

The third step is to set back the first decision to the value it had for the baseline scenario and identify and flip the second decision in the baseline scenario while holding all other decisions to their baseline values. This, likewise, should produce a third scenario which is minimally different from the baseline scenario.

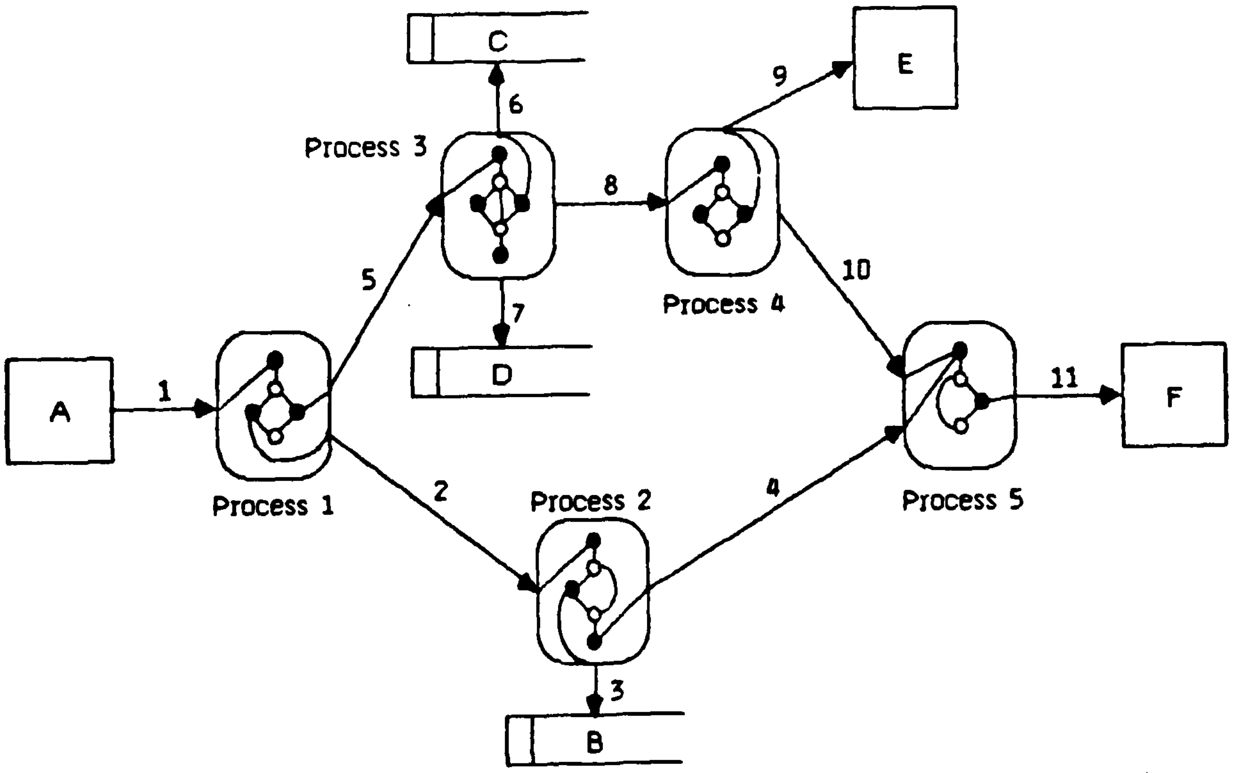
This procedure continues until one has gone through every decision and has flipped it from the baseline value while holding the other decisions to their original baseline values.

Since v(DFD) = pi + 1, if, for example, v = 7, there are 6 such decisions which are flipped - resulting in 6 scenarios that differ from the baseline scenario. These plus the baseline make up a set of 7 basis scenarios.

Since the selection of the baseline scenario is somewhat arbitrary there is not necessarily "the" right set of scenarios for a DFD. The application of this baseline method will, nonetheless, generate a set of scenarios such that:

1) v(DFD) distinct independent scenarios will be generated
2) every flow in the DFD will be traversed

For example:



Suppose we are given a component such as the one depicted above. Note that a black dot reduced graph of the process logic is included, and each flow is labelled.

Its complexity is pi + 1 = 6 + 1 = 7.

Now let's follow the steps detailed above to generate scenarios.

1) Choose a baseline. Keep in mind that this ideally performs the major full function provided in the DFD. If possible, choose a scenario that intersects a maximal number of decisions. Let's choose 1,5,6,8,10,11.

2) Now flip the first decision along the baseline. In this case, we again somewhat arbitrarily choose a scenario, as there is more than one scenario equally different from the baseline after flipping the first decision. Let's choose 1,2,3,4,11.

3) Now we'll need to flip the decision in process 2. We get 1,2,4,11.

4) Now return the first decision to the baseline and flip the second decision along the baseline - 1,5,7,8,10,11.

5) We have one more scenario to generate with the second decision different from the baseline - 1,5,8,10,11.

6) Now we flip the third decision along the baseline, generating 1,5,6,8,9.

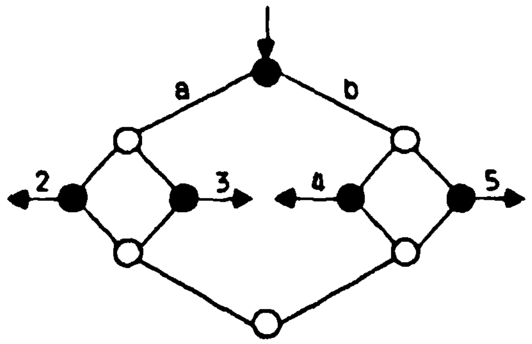
7) We have one more decision to flip along the baseline. We get 1,5,6,8,10.

Since we have completely flipped every decision, the procedure is completed. Notice that every flow has been traversed, and we have generated 7 independent scenarios, which matches the data flow complexity v(DFD).

BASELINE FOR LOGIC WITH PARALLEL FLOWS

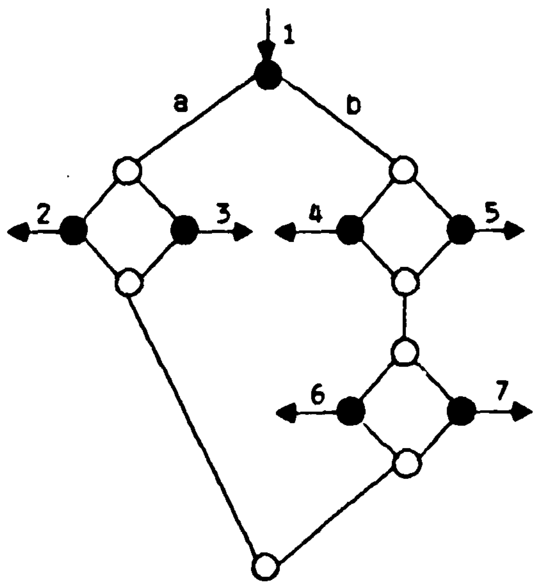
Upon occasion we find DFDs with parallel flows. We will now examine how to apply the baseline method to process logic with such flows. The method is quite similar to that for sequential flows.

Suppose we are given the graph below:



Note that flows a and b can operate in parallel. Let's determine complexity - there are two decisions, pi = 2, so v(G) = 3. First pick a baseline - 1,2,4. Note that flows along both sides of the graph are traversed due to the parallel structure. Now we'll flip one of the decisions - 1,3,4. We return to the baseline and flip the other decision - 1,2,5. We have generated 3 linearly independent scenarios which span the set of all scenarios.

Let's look at a slightly more complicated graph -



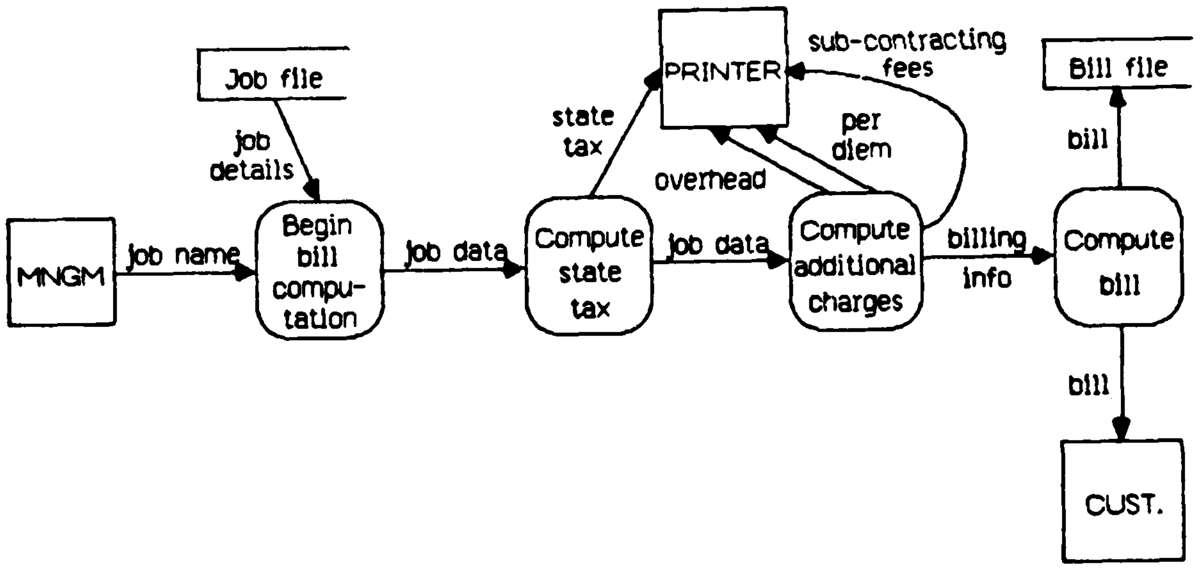
v(G) = 3 + 1 = 4

Let's choose 1,2,4,6 as a baseline. Flip a decision - 1,3,4,6. Now return to the baseline and flip a decision along another a parallel branch - 1,2,5,6. Flip the next decision down this branch - 1,2,4,7. Now we have exercised each decision along every branch once. Furthermore, we have a set of v(G) linearly independent scenarios.

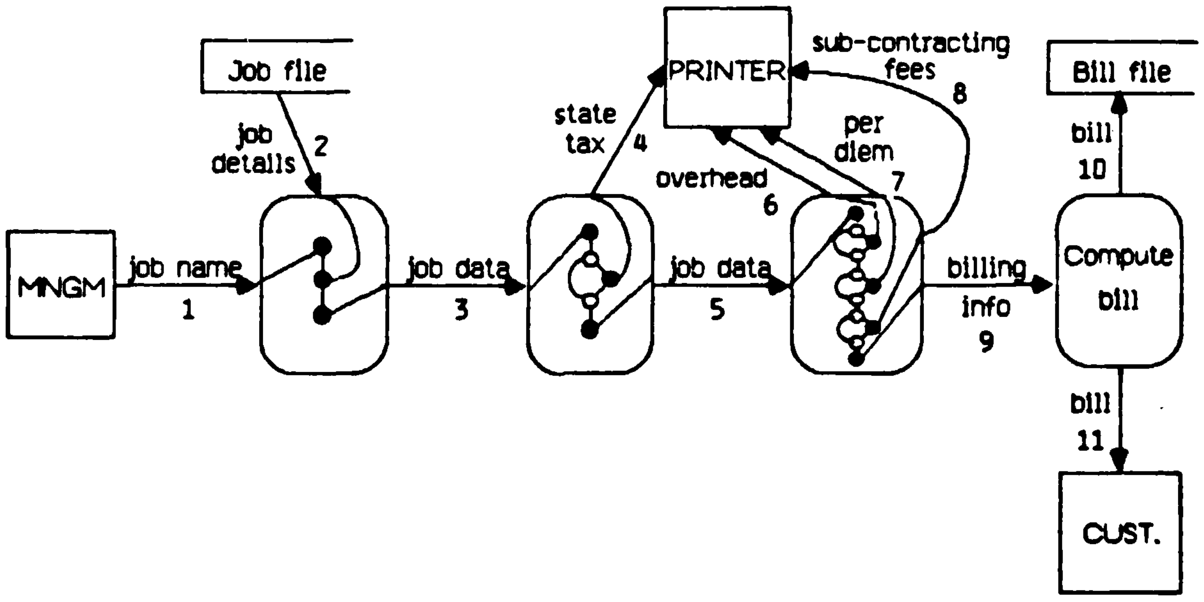
XVI. AN ANALYSIS TOOL

We have developed a Specifications Complexity Analysis Tool (SCAT) which performs several useful functions to help us in our analysis of DFDs. Scenarios are input as matrix rows; each column represents a data flow. After the test set is analyzed, other scenarios and data flows can be added - the user can then check for changes in complexity, basis scenarios, etc. Editing features, allowing scenarios to be modified, rows and columns to be added, and old files to be changed are included. The matrices can be saved and retrieved from disk. The last time accessed and number of updates are recorded.

Several outputs are available; the user can choose which are required. The DFD below will be used to illustrate the tool. Scenarios, generated from the process logic, are input into the tool. Sample outputs with explanatory notes follow.



```
Begin Bill Computation

    Receive Job Name
    Load Job Details
    Compute labor/materials costs
    Pass Job Data to Compute State Tax

Compute State Tax
    If job in-state then,
       Compute state taxes
       Output State Tax amount to PRINTER
    Pass Job Data to Compute Additional Charges

Compute Additional Charges
    If job in-house then,
       Compute overhead
       Output Overhead to PRINTER
    If job on-site then,
       Compute per-diem
       Output Per-Diem to PRINTER
    If job subcontracted then,
       Compute subcontracting fees
       Output Subcontracting Fees to PRINTER
    Pass Billing Info to Compute Bill

Compute Bill

    Compute bill
    Send Bill to Bill File
    Send Bill to CUST.
```



```
Test Set: New Employee          Update: 0
User: McCabe & Associates       Date: 1/7/83
                                Last Accessed:


Original Matrix

        Data Flows
         1  2  3  4  5  6  7  8  9 10 11

T  1     1  1  1  1  1  1  1  1  1  1  1
e  2     1  1  1  1  1  0  1  1  1  1  1
s  3     1  1  1  1  1  0  0  1  1  1  1
t  4     1  1  1  1  1  0  0  0  1  1  1
   5     1  1  1  1  1  0  1  0  1  1  1
S  6     1  1  1  1  1  1  0  1  1  1  1
c  7     1  1  1  1  1  1  0  0  1  1  1
e  8     1  1  1  1  1  1  1  0  1  1  1
n  9     1  1  1  1  1  1  1  0  1  1  1
a 10     1  1  1  0  1  1  1  1  1  1  1
r 11     1  1  1  0  1  0  1  1  1  1  1
i 12     1  1  1  0  1  0  0  1  1  1  1
o 13     1  1  1  0  1  0  0  0  1  1  1
s 14     1  1  1  0  1  0  1  0  1  1  1
  15     1  1  1  0  1  1  0  1  1  1  1
  16     1  1  1  0  1  1  0  0  1  1  1
  17     1  1  1  0  1  1  1  0  1  1  1
  18     1  1  1  1  1  1  0  1  1  1  1
```

Original Matrix - This is an output of the scenarios in the order originally entered. Optionally the complexity and basis scenarios are listed. For example, the 15th scenario is composed of flows 1,2,3,5,6,8,9,10 and 11. The complexity of the matrix is 5, and the basis scenarios are 13,14,16,4, and 12.

```
External Complexity = 5

Linearly Independent Test Scenarios: 13, 14, 16, 4, 12,
Test Set: New Employee          Update: 0
User: McCabe & Associates       Date: 1/7/83
                                Last Accessed:
```

Ordered Matrix

		Data Flows										
		1	2	3	4	5	6	7	8	9	10	11
T	13	1	1	1	0	1	0	0	0	1	1	1
e	14	1	1	1	0	1	0	1	0	1	1	1
s	16	1	1	1	0	1	1	0	0	1	1	1
t	4	1	1	1	1	1	0	0	0	1	1	1
	12	1	1	1	0	1	0	0	1	1	1	1
S	3	1	1	1	1	1	0	0	1	1	1	1
c	7	1	1	1	1	1	1	0	0	1	1	1
e	5	1	1	1	1	1	0	1	0	1	1	1
n	17	1	1	1	0	1	1	1	0	1	1	1
a	15	1	1	1	0	1	1	0	1	1	1	1
r	11	1	1	1	0	1	0	1	1	1	1	1
i	9	1	1	1	1	1	1	1	0	1	1	1
o	8	1	1	1	1	1	1	1	0	1	1	1
s	6	1	1	1	1	1	1	0	1	1	1	1
	10	1	1	1	0	1	1	1	1	1	1	1
	2	1	1	1	1	1	0	1	1	1	1	1
	18	1	1	1	1	1	1	0	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1

Ordered Matrix - This matrix shows the scenarios ordered so that more basic scenarios are listed first. Scenarios are first ordered by number of dimensions spanned, then by magnitude. As an example, the tool placed scenario 13 first. It covers only seven flows, whereas all the other scenarios cover more. If a scenario with one flow across three data flows and more than one flow across one other were added, this would appear between scenario 13 and scenario 14.

```
Test Set: New Employee             Update: 0
User: McCabe & Associates          Date: 1/7/87
                                   Last Accessed:



Scenario Composition

         Data Flows
          1  2  3  4  5  6  7  8  9 10 11

T 13      1  0  0  0  0  0  0  0  0  0  0
e 14      0  1  0  0  0  0  0  0  0  0  0
s 16      0  0  1  0  0  0  0  0  0  0  0
t  4      0  0  0  1  0  0  0  0  0  0  0
  12      0  0  0  0  1  0  0  0  0  0  0
S  3     -1  0  0  1  1  0  0  0  0  0  0
c  7     -1  0  1  1  0  0  0  0  0  0  0
e  5     -1  1  0  1  0  0  0  0  0  0  0
n 17     -1  1  1  0  0  0  0  0  0  0  0
a 15     -1  0  1  0  1  0  0  0  0  0  0
r 11     -1  1  0  0  1  0  0  0  0  0  0
i  9     -2  1  1  1  0  0  0  0  0  0  0
o  8     -2  1  1  1  0  0  0  0  0  0  0
s  6     -2  0  1  1  1  0  0  0  0  0  0
  10     -2  1  1  0  1  0  0  0  0  0  0
   2     -2  1  0  1  1  0  0  0  0  0  0
  18     -2  0  1  1  1  0  0  0  0  0  0
   1     -3  1  1  1  1  0  0  0  0  0  0
```

Scenario Compositon - This matrix indicates the composition of scenarios with respect to a basis set which spans it. Basis vectors will have a 1 in one column; all other columns will be zero. For example, scenario 4 is a basis scenario. The composition of other scenarios is found by tracing column entries up to a basis scenario with one in that column. For example, scenario 3 is the same as negative scenario 13 plus scenario 4 plus scenario 12. Scenario 10 is the same as negative two times scenario 13 plus scenario 14 plus scenario 16 plus scenario 12.

```
Test Set: New Employee           Update: 0
User: McCabe & Associates        Date: 1/7/83
                                 Last Accessed:


Flow Frequency Count

 1 #################################################################################################### 18
 2 #################################################################################################### 18
 3 #################################################################################################### 18
 4 ########################################################                                             10
 5 #################################################################################################### 18
 6 ########################################################                                             10
 7 ##################################################                                                   9
 8 ##################################################                                                   9
 9 #################################################################################################### 18
10 #################################################################################################### 18
11 #################################################################################################### 18

# = .18
```

Flow Frequency Count - This chart shows the number of times each flow was traversed. It is useful in pointing out how the system was stressed, and readily indicates any flows which haven't been covered. For example, flow 1 has been traversed eighteen times, flow 4 ten times. In this example, all of the flows have been traversed.

```
Test Set: New Employee          Update: 0
User: McCabe & Associates       Date: 1/7/83
                                Last Accessed:


Suggested Order

        Data Flows
        1 2 3 4 5 6 7 8 9 10 11

T 13    1 1 1 0 1 0 0 0 1 1  1
e 14    1 1 1 0 1 0 1 0 1 1  1
s 16    1 1 1 0 1 1 0 0 1 1  1
t  4    1 1 1 1 1 0 0 0 1 1  1
  12    1 1 1 0 1 0 0 1 1 1  1

S  1    1 1 1 1 1 1 1 1 1 1  1

c 18    1 1 1 1 1 1 0 1 1 1  1
e  2    1 1 1 1 1 0 1 1 1 1  1
n 10    1 1 1 0 1 1 1 1 1 1  1
a  6    1 1 1 1 1 1 0 1 1 1  1
r  8    1 1 1 1 1 1 1 0 1 1  1
i  9    1 1 1 1 1 1 1 0 1 1  1

o 11    1 1 1 0 1 0 1 1 1 1  1
s 15    1 1 1 0 1 1 0 1 1 1  1
  17    1 1 1 0 1 1 1 0 1 1  1
   5    1 1 1 1 1 0 1 0 1 1  1
   7    1 1 1 1 1 1 0 0 1 1  1
   3    1 1 1 1 1 0 0 1 1 1  1
```

Suggested Order - This list separates tests into groups of equal quality. Scenarios of high quality, which will appear in an upper group, should be tested before those of lower quality - those at the lower end of the list. Scenarios of highest quality are basis scenarios, followed by scenarios which combine several basis scenarios, according to the chart below:

"Goodness" of Test Scenario

(ranked in descending order)

Linearly independent test scenario
Multiple of v(G) basis scenarios
Multiple of v(G)-1 basis scenarios
.
.
.
Multiple of two basis scenarios
Multiple of one basis scenario
Repetition of a basis scenario

For example, the upper block, scenarios 13,14,16,4, and 12, is composed of the basis scenarios. These should be tested first. Scenario 1 is composed of 5 basis scenarios. If more than 5 (i.e. v(G)) scenarios are to be tested, scenario 1 should be tested next.

This tool is helpful in the following several ways:

- When a new scenario is added to the test bed, its relative independency is computed.

- The external complexity of a test matrix (made up of scenario rows) is quickly computed; this external complexity can then be checked against a DFD's internal complexity ($\Sigma pi + 1$) to see if there are independent scenarios yet untested that should be added to the matrix.

- The frequency count analysis is useful to highlight a flow that has not been tested. This analysis is also typically used to drive stress testing.

- The suggested order output (page 70) helps to identify the sets of tests to be run first. It then ranks the remaining tests into groups of equal priority of testing, to be run as the testing budget and schedule permit.

XVII. RELATIONSHIP OF TEST SCENARIOS WITH CAUSE-EFFECT FORM OF SPECIFICATIONS

While data flow diagrams provide an easily understood visual approach to specifications, they have a weakness as a specification document. The DFD is only a picture, and thus doesn't show an explicit list of exactly what the system provides to the user. Such a list form of specifications is sometimes called a cause and effect pair list[12]. The cause and effect list is an explicit, numbered list of external stimuli to the system and their associated effects. It provides a more explicit mechanism to assure that the implementation is a full one meeting all the cause and effect relationships.

Our emphasis up to this point has been to identify and quantify the scenarios within a DFD for purposes of testing. Our orientation is to quantify the inherent complexity within a DFD and have the quantification provide a method to derive the distinct and independent scenarios - once again from a testing perspective. We would like, however, to change the perspective for a moment to specification instead of testing.

The troublesome issue in turning a DFD into a cause and effect list is the unknown number and identity of the cause and effect pairs existing within a DFD. What we have been calling test scenarios are indeed cause and effect pairs. By definition each is stimulated by an external entity and terminated in a non-process; the external entity is the cause, the non-process termination is the effect. One can thus apply our testing reasoning to the often error prone process of turning a DFD picture into a specific list of cause and effect pairs.

The DFD complexity, in terms of rank of the test matrix or pi+1, quantifies the number of independent cause and effect pairs. This therefore gives a lower bound to the number of cause and effect pairs that must be listed; the analysis we describe with the DFD tool will likewise indicate when the cause and effect pairs are redundant in the sense of being linear combinations of basis pairs. Our ideas of basis test scenarios may be translated into the concept of basis capabilities in terms of cause and effect pairs. A notion of redundancy of test data is directly interpretable to redundancy of cause and effect specifications.

All of this is not surprising. A rigorous set of test data for a system can be thought of as a specification for its behavior. There is an intimate relationship between the test data for a system and the specifications. If a capability is specified there should be test data that demonstrates it; any set of test data must in turn demonstrate a specified capability. This intrinsic relationship between a system's specification and its test data is intuitively obvious, the difficulty is putting it into practice in a rigorous way. While our focus has been on rigorizing the formulation of test data for a system, the same mathematical quantification can indeed rigorize specifications.

Every idea and procedure described in this book for a testing application can be interpreted within a specifications framework. If one is interested in specifications and not testing, all he need do is read the book and substitute "specification" for "test", and "cause and effect pair" for "scenario".

Now let's look at the situation from the reverse point of view - we are given a listing of all the input pairs to a system and the outputs they invoke. This listing is obviously a system specification. Drawing upon the analogy between scenarios and cause and effect pairs, we see that a cause and effect pair specification implies that certain scenarios exist within a system. In fact, these are the only scenarios which must exist in a system which corresponds to the specification. Thus, we can say that the complexity of any system satisfying the specification should be equal to the complexity of the cause and effect pair specification! Furthermore, a system with complexity greater than the specification complexity is cluttered and repetitive; one with a lesser complexity cannot satisfy the specification.

XVIII. OVERALL EXAMPLE

An example will be walked through to illustrate the operational steps described in the various chapters. The chapters referred to in each step are listed at the end of each step.

Figure 1 shows a high level specification in DFD form. DFDs are leveled documents - each one can be broken down and shown at a lower level. For example, Figure 2 shows a breakdown of process 1 and the associated process logic.

The operational steps follow:

1) Identify the components. Note that Figure 1 is composed of four components.

1) process received payments	(process 1)
2) process and ship orders	(processes 2 and 3)
3) process employee changes	(process 4)
4) handle payroll and accounting	(processes 5 and 6)

(Chapter IX : Examination of Components)

2) Process logic should be associated with each DFD. Figure 3 shows the DFD from Figure 2 with its process logic filled in. (Chapter VII : Internal Process Logic)

3) Now perform a black dot reduction. Figure 3 generates Figure 4. (Chapter VII : Internal Process Logic)

4) Number the flows on the reduced graph, as shown in Figure 4.

5) Compute the complexity. Since the DFD in Figure 4 contains just one component, the complexity of the DFD = pi + 1 = 5 + 1 = 6. Therefore there are five scenarios plus the baseline to be generated. (Chapter X : The Relationship Between the Complexity of a Component and the Complexity of its Internal Processes)

6) Generate scenarios. (Chapter XV : Identifying Scenarios - The Baseline Method)

6.1) Choose a baseline. Let's choose 1,2,4,5,9,10,15.
6.2) Flip the decision in process 1.1, giving the scenario 1,2,3,11.
6.3) Now return to the baseline and flip the decision in process 1.2. We get 1,2,4,5,6,7,8.
6.4) Once again, return to the baseline. Flip the first decision in process 1.4 to give us 1,2,4,5,9,10,13.
6.5) Using the scenario we just generated, flip the last decision resulting in 1,2,4,5,9,10,14.

6.6) Return to the baseline, this time flipping the second decision in process 1.4 which gives us 1,2,4,5,9,10,12.

7) Process these scenarios with the Specifications Complexity Analysis Tool, which produces the outputs like those of Figure 5. (Chapter XVI : An Analysis Tool)

This discussion has focussed on just one component. When executing the scenarios within the other components, the Law of Conservation of Data should be applied (Chapter XIV : Integration of Components). The effect will be to use "live" test data that spans the four components. This will result in high level integration across components.

An additional example, using steps two and three, is shown in Figures 6-8.

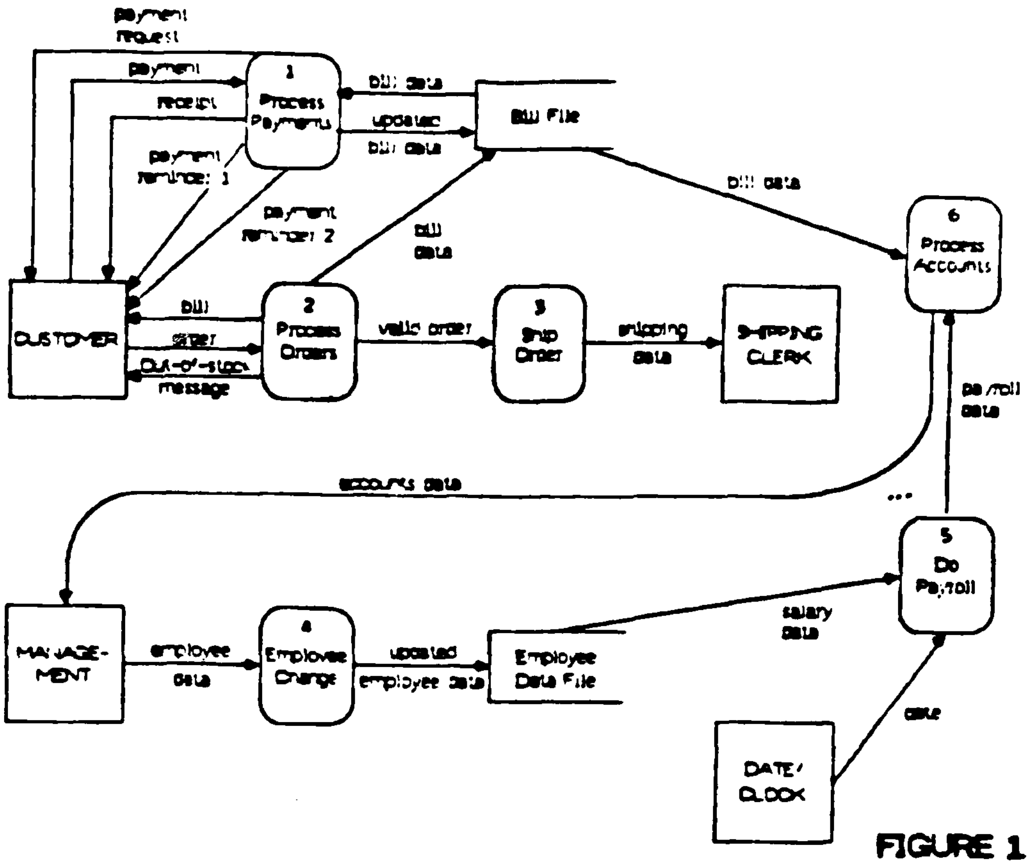


FIGURE 1

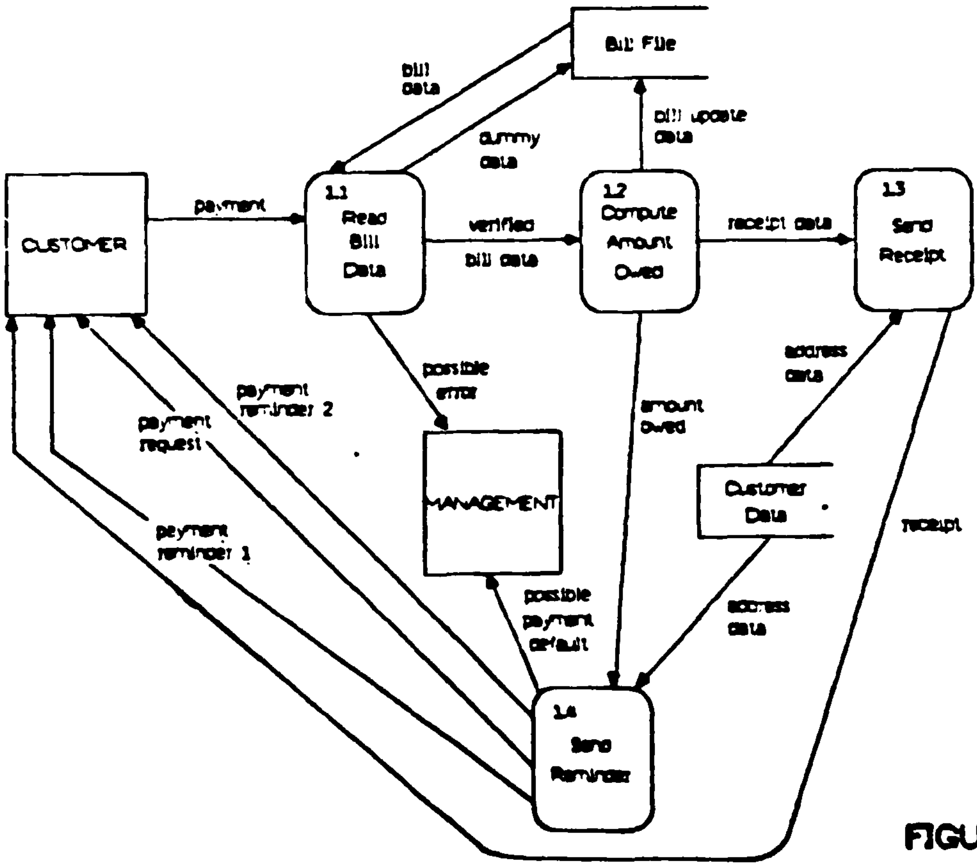


FIGURE 2

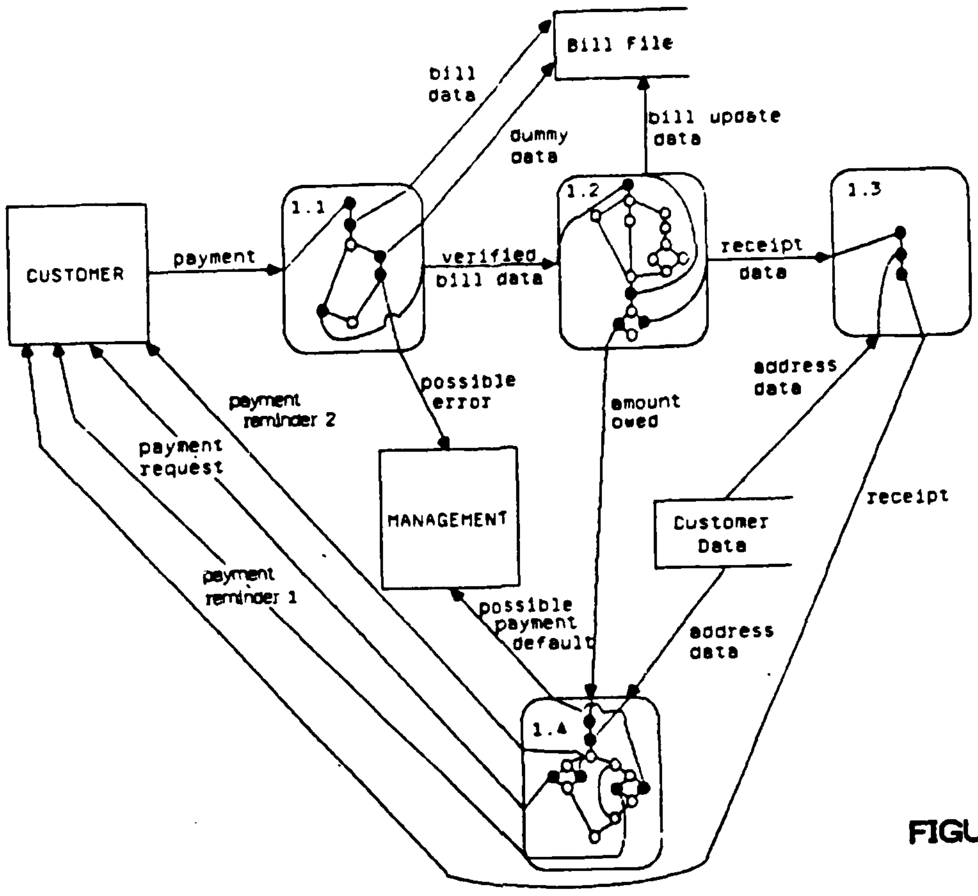


FIGURE 3

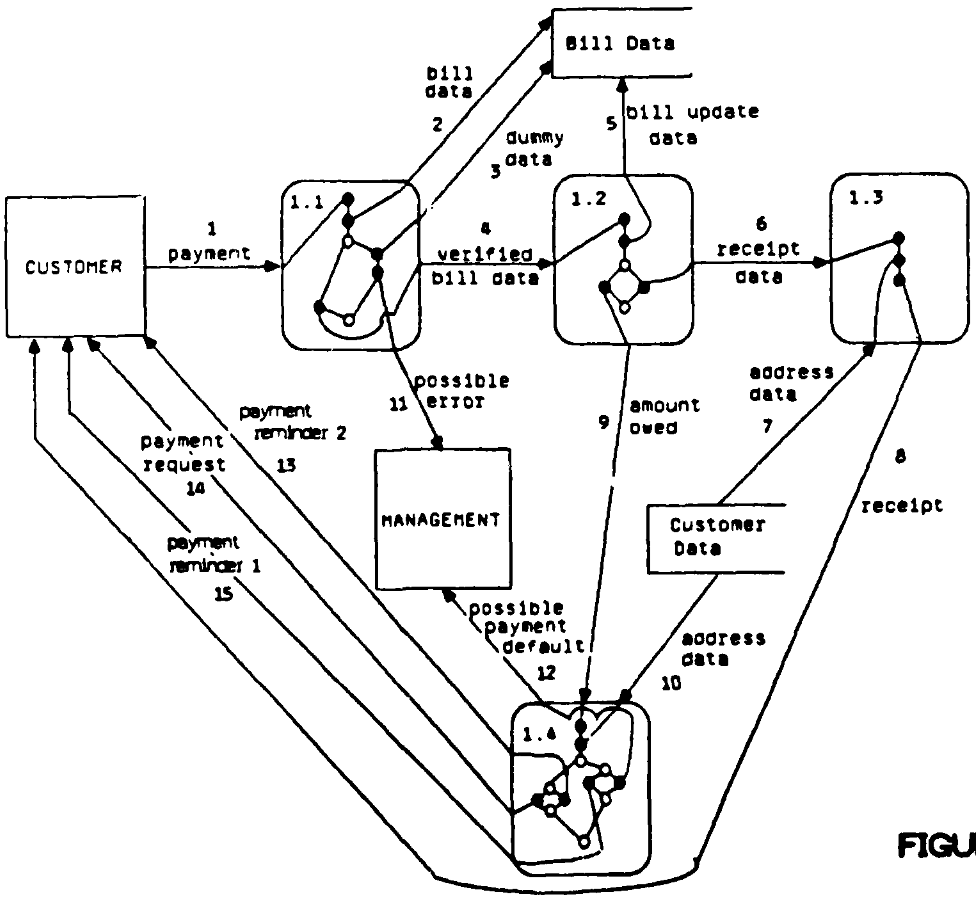


FIGURE 4

```
Original Matrix - This is an output of the scenarios in the order originally entered.

       Data Flows
        1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

T 1     1  1  0  1  1  0  0  0  1  1  0  0  0  0  1
e 2     1  1  1  0  0  0  0  0  0  0  1  0  0  0  0
s 3     1  1  0  1  1  1  1  1  0  0  0  0  0  0  0
t 4     1  1  0  1  1  0  0  0  1  1  0  0  1  0  0
  5     1  1  0  1  1  0  0  0  1  1  0  0  0  1  0
S 6     1  1  0  1  1  0  0  0  1  1  0  1  0  0  0
```

```
External Complexity = 6

Linearly Independent Test Scenarios: 2, 1, 3, 4, 5, 6,
```

```
Scenario Composition - This matrix indicates the composition of scenarios with respect to a basis set which spans it.

       Data Flows
        1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

T 2     1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
e 1     0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
s 3     0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
t 4     0  0  0  1  0  0  0  0  0  0  0  0  0  0  0
  5     0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
S 6     0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
```

```
Flow Frequency Count - This chart shows the number of times each
flow was traversed

 1 ####################################################################################################  6
 2 ####################################################################################################  6
 3 #################                                                                                     1
 4 ###################################################################################                   5
 5 ###################################################################################                   5
 6 #################                                                                                     1
 7 #################                                                                                     1
 8 #################                                                                                     1
 9 ###################################################################                                   4
10 ###################################################################                                   4
11 #################                                                                                     1
12 #################                                                                                     1
13 #################                                                                                     1
14 #################                                                                                     1
15 #################                                                                                     1

# = .06
```

FIGURE 5

Process 1.1 READ BILL DATA

```
Get payment from customer
Read bill data from bill file
If no bill data
      Then,
            Send dummy data to bill file
            Send "possible error" message to management
      Otherwise
            Pass verified bill data to Compute Amount Owed
```

Process 1.2 COMPUTE AMOUNT OWED

```
Compute elapsed time
Select the case which applies:
        Elapsed time , 30 days
                  Subtract payment from amount owed
        Elapsed time ¢ 30 days, , 90 days
                  Subtract payment from amount owed
                  Add 5% interest to new amount owed
        Elapsed time ¢ 90 days
                  Add 5% interest to amount owed
                  Subtract payment from amount owed
                  If amount owed ¢ $1000,
                              Then,
                                    Increase amount owed by $200
Send bill update data to bill file
If new amount owed = $0,
        Then
             Pass receipt data to send receipt
        Otherwise
             Pass amount owed to send reminder
```

Process 1.3 SEND RECEIPT

```
Read address data from customer data
Send receipt to customer
```

Process 1.4 SEND REMINDER

```
Read address data from customer data
If amount owed ¢ $1000
        Then,
              If elapsed time ¢ 90 days
                     Then,
                           If elapsed time , 120 days
                                  Send payment reminder 1 to
                                  customer
                           Otherwise,
                                  Send "possible payment
                                  default" message to
                                  management
Otherwise,
        If elapsed time ¢ 120 days,
              Then,
                    Send payment reminder 2 to customer
              Otherwise,
                    Send request for payment to customer
```

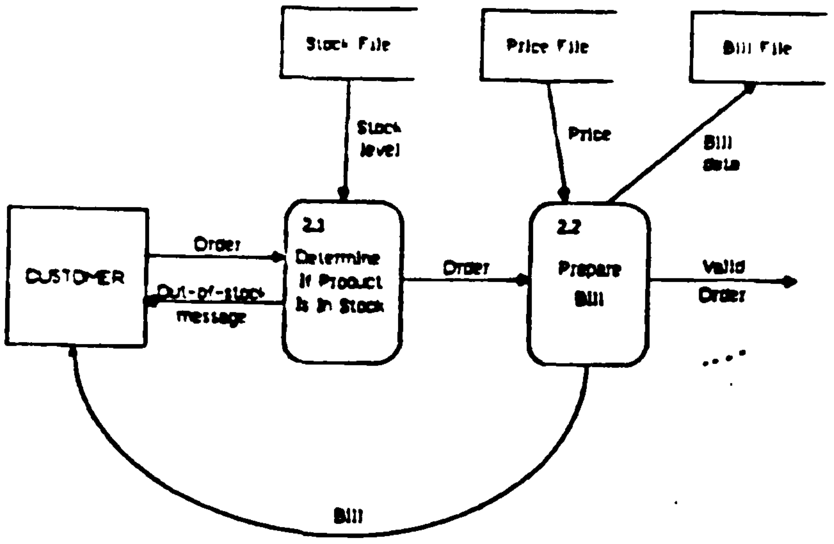


FIGURE 6

```
Process 2.1          DETERMINE IF PRODUCT IS IN STOCK

        Get order from customer
        Read stock level from stock file
        If stock level = 0
                 Then,
                      Send out-of-stock message to customer
                 Otherwise,
                      Send order to Prepare Bill


Process 2.2          PREPARE BILL

        For each item in order
              Read price from price file
        If total amount > $100
                  Then,
                       Add 10% shipping charge
        Send bill to customer
        Send valid order to Ship Order
```

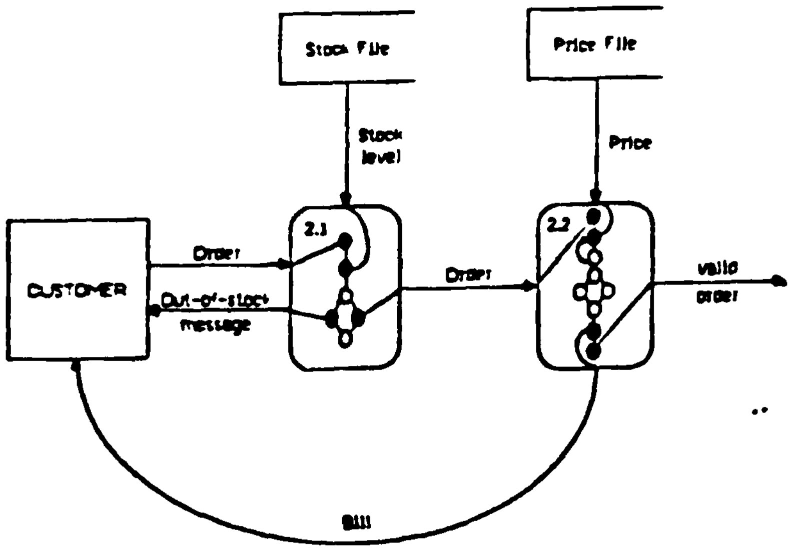


FIGURE 7

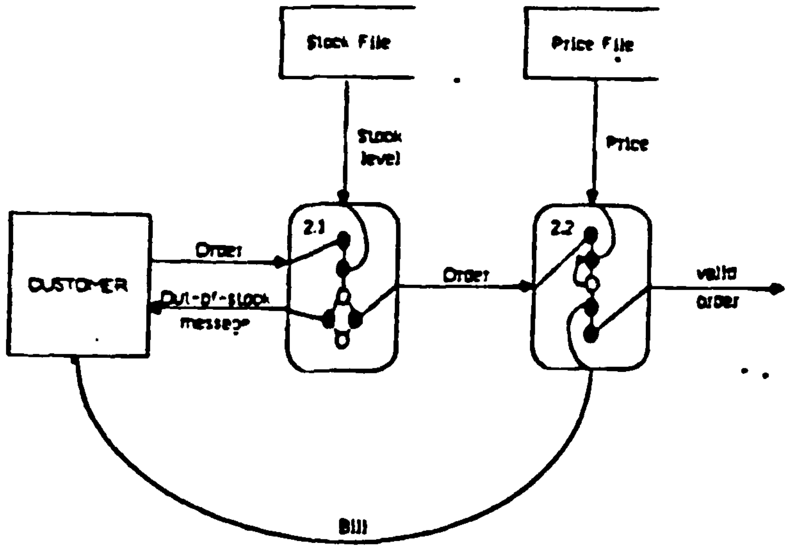


FIGURE 8

REFERENCES

1. Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment", Datamation, May, 1973.

2. Boehm, Barry W., Software Engineering Economics, Prentice-Hall, 1981.

3. DeMarco, Tom, Structured Analysis and System Specification, Prentice-Hall, 1979.

4. McCabe, Thomas J., "A Complexity Measure", IEEE Transactions on Software Engineering, 1976.

5. Berg, Claude, The Theory of Graphs and Its Applications, John Wiley and Sons, Inc., 1958.

6. Legard, H. and Marcotty, M., "A Generalogy of Control Structures", Commun. Assoc. Computing Machines, November, 1975.

7. McCabe, Thomas J., Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, NBS Special Publication 500-599, December, 1982.

8. Cullen, Charles, Matrices and Linear Transformations, Addison-Wesley Publishing Co., 1967.

9. See 8, above.

10. Jane, C. and Sarson, T., Structured Systems Analysis: Tools and Techniques, Improved System Technologies, Inc., 1977.

11. See 7, above.

12. Myers, G.J., The Art of Software Testing, John Wiley and Sons, 1979.

FIL